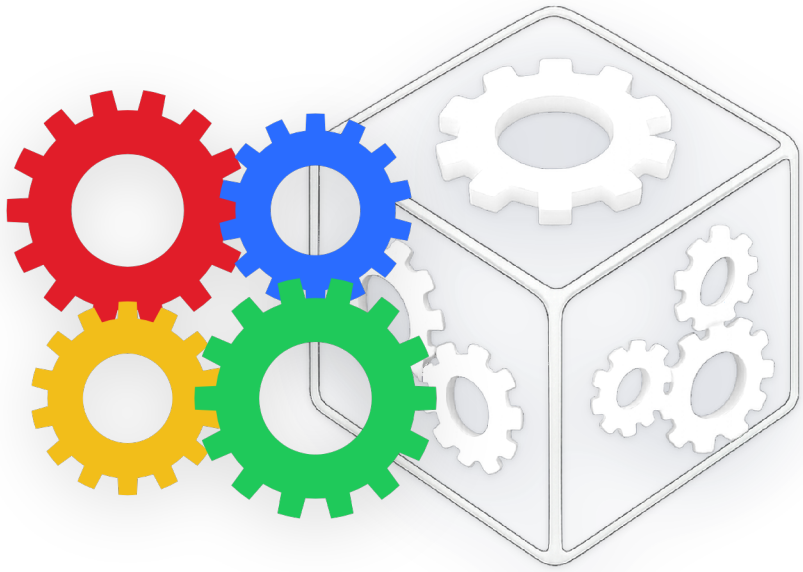


TECHNICAL SEO · AUDITS · TOOLING

Technical SEO for Digital Marketing

A Guided Technical-SEO Course with Executed Audit Tools

2026 Edition · Luke McLaughlin

Technical SEO for Digital Marketing

2026 EDITION

Copyright © 2026 Luke McLaughlin. All rights reserved.

The moral right of the author to be identified as the author of this work has been asserted.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise) without the prior written permission of the author, except for brief quotations embodied in reviews and other non-commercial uses permitted by copyright law.

Written, designed, and **independently published by Luke McLaughlin**.

This book is intended for educational purposes. While every effort has been made to ensure the accuracy of its contents, the author assumes no responsibility for errors or omissions, or for any outcome arising from the use of the material herein.

Luke McLaughlin · Author & Publisher

lukekevinmclaughlin@gmail.com · lukekevinmclaughlin.com

First edition, 2026

Preface

This edition turns technical SEO into a guided, hands-on course: 14 chapters, full concept lessons, worked traces, visual models, checkpoints, quizzes and executed practice projects that crawl, parse and audit the way a real search engine does — from robots.txt and XML sitemaps to status codes, on-page signals, Core Web Vitals, structured data, hreflang and the new world of AI answer engines.

The book keeps a strict code-and-output rhythm: every project is a small, deterministic Python tool that runs against a worked example — a robots.txt string, a sitemap, an HTML snippet, a redirect chain, a JSON-LD block — prints a verdict, and shows you the exact output. You read the audit logic, predict the result, then change one detail and watch what moves. In production you point the same logic at a live URL; here you point it at a fixture so the lesson is reproducible.

Work through the chapters in order for a complete path from how crawling and indexing actually work, through on-page and content signals, performance and structured data, to international, off-page, local and AI-era optimization — finishing with an end-to-end technical audit you can run on any site. Each chapter opens with the concept, then builds the tools, so the tactics never feel like a checklist you memorize: they are habits you can execute.

About the Author



Luke McLaughlin is an independent author and publisher who grew up across several countries, an upbringing that left him endlessly curious and gave him a lifetime of overlapping passions. His path has wound through molecular biology, writing, digital growth, and software and artificial intelligence, including building *Codemonkey AI*, a desktop tutor that teaches people to code. The thread running through all of it is a love of taking something that looks complicated and uncovering the simple ideas underneath.

That is also his philosophy of learning. He treats every subject as a series of **building blocks**, each one resting on the one before it. Master the blocks in order and no subject is truly out of reach: give a little time, effort, and patience to each step, and even the most daunting topic gives way. This book was written in exactly that spirit.

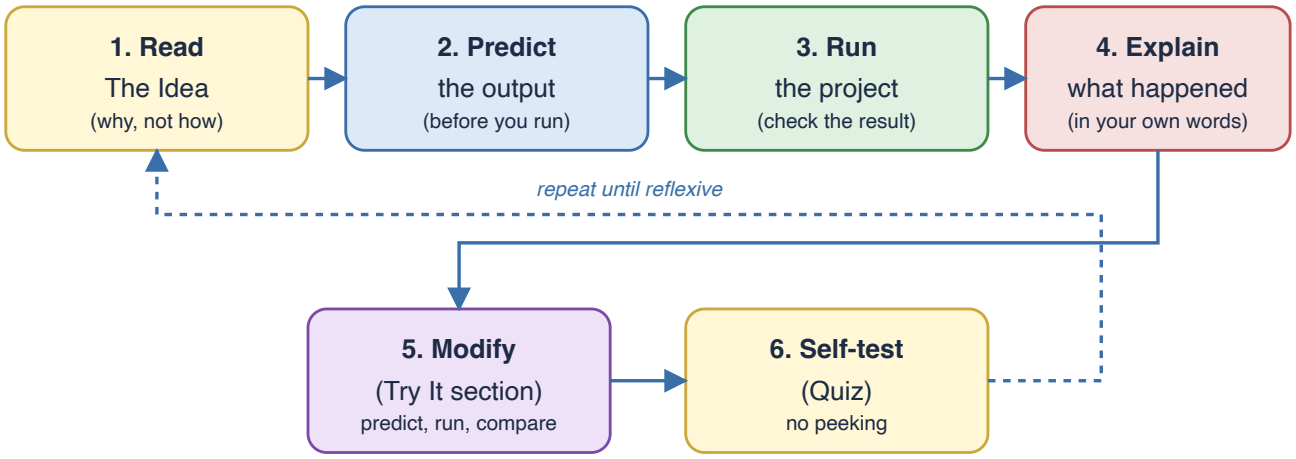
> *"Every journey of a thousand miles begins with a single step."*

lukekevinmclaughlin@gmail.com · lukekevinmclaughlin.com

How to Use This Book

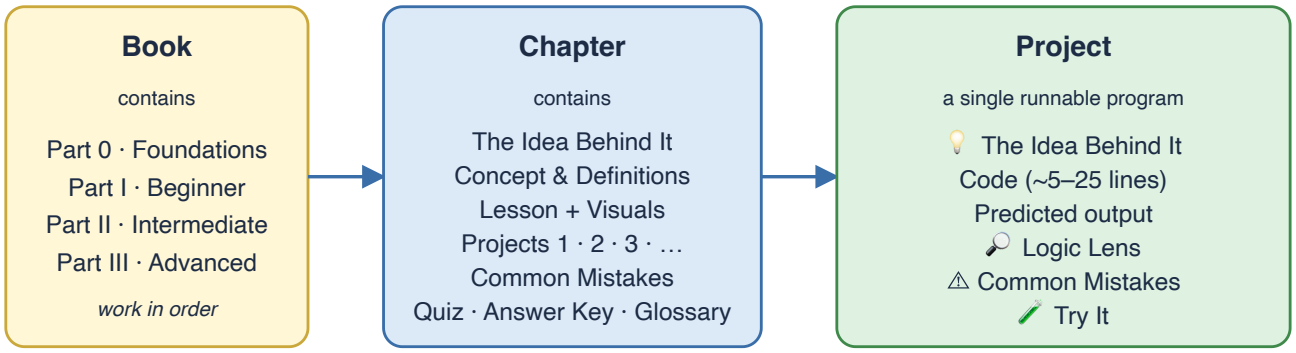
Every chapter in this book is built around one promise: a complete novice with only basic arithmetic should finish able to read, write, debug, and reason about real programs. To get there, the book follows the **essentials educational method** — a small, predictable rhythm repeated until the moves become reflexive.

The essentials method, in six steps



Every project is a turn of this loop. Run it deliberately five times per chapter and the patterns wear in.

The structure that supports the loop



Parts contain chapters; chapters contain ideas, lessons, and projects; every project contains the same internal rhythm — concept, code, prediction, mistake list, experiment.

The signposts you will see

> 💡 ***The Idea Behind It*** – **The why before the how.** Every chapter and every project opens with the thinking that produced the code. Read this slowly. If the idea is clear, the code will be obvious; if the idea is fuzzy, the code will feel like magic.

📖 ***Concept & Definitions*** – **The vocabulary in plain English.** A handful of terms a learner needs before reading the lesson. Each term is paired with a one-line definition and an example.

🧭 ***Chapter Skill Map & Roadmap*** – **Where you are and what you are building.** A short orientation showing what skill the chapter adds, what it assumes from earlier chapters, and how the projects step through the new idea.

🛠️ ***Lesson Walkthrough*** – **The idea applied, step by step.** Short sections with worked code, tables that compare options, and visuals that turn abstract structure into something you can see.

Project N → code + output – **The complete, runnable program with the exact output to check against.** Predict the output *before* running; matching the prediction proves you understood the idea, not just typed the code.

🔎 ***Logic Lens*** – **What this line really does.** A short stack of explanations that re-reads the project as a sequence of small decisions — useful when a project surprises you.

⚠️ ***Common Mistakes*** – **Traps most learners fall into.** Each entry names the mistake and the one-line fix; reading them before you start a project saves an hour of debugging.

🧪 ***Try It*** – **Small modifications that turn the example into your own experiment.** Change one thing, predict, run, compare. This is where ideas become reflexes.

📝 ***Quiz & Answer Key*** – **A short self-check at the end of every chapter.** Full answers follow in the chapter's answer key — every quiz answer is explained, not just marked.

📚 ***Glossary*** – **The chapter's vocabulary in one place.** Use it as you read, and again later when you come back to refresh the chapter.

Working the book the essentials way

1. **Read each chapter's Idea before any code.** Programming is a translation skill, not a typing skill. The *idea* is what you are translating from; the code is what you are translating to.
2. **Predict every project's output before running it.** The point is not to be right — the point is to make your mental model concrete enough to be wrong. Mismatches are the most useful learning events in the book.
3. **Do every Try It.** They are short on purpose. Changing one thing, predicting, and re-running is how the patterns become reflexes; reading without modifying does not transfer to your own programs.
4. **Take the quiz with the book closed.** If you peek, you are reading; if you predict, you are learning. Use the answer key after — the explanations are written to teach, not just to mark.
5. **Treat every mistake as data.** When the code does not match the prediction, write down what surprised you in one sentence before you fix it. Those sentences become your personal debugger over time.
6. **Move forward only when you can explain the chapter aloud.** If you can tell a friend in two minutes what the idea was and why the projects show it, you are ready. If not, re-read the Idea, re-run two projects, and try again — a day of consolidation now saves a week later.

> *Programming rewards consistency more than intensity. Thirty focused minutes a day, applied through the loop above, will carry you remarkably far. The book is structured so that every chapter you finish is one you understand — never finish a chapter you cannot explain. That is the whole method.*

Contents

Part I – Beginner Practice

Chapter 1 – How Search Engines Work 10
Chapter 2 – robots.txt and Crawl Directives 136
Chapter 3 – XML Sitemaps and URL Discovery 304
Chapter 4 – HTTP Status Codes and Redirects 468

Part II – Intermediate Skills

Chapter 5 – On-Page Signals and the HTML Audit 631
Chapter 6 – Keyword Research and Intent as Data 794
Chapter 7 – Canonicalization, Duplicate and Thin Content 941
Chapter 8 – Site Speed and Core Web Vitals 1102
Chapter 9 – Structured Data and Rich Results 1257

Part III – Advanced Mastery

Chapter 10 – International SEO and hreflang 1410
Chapter 11 – Off-Page, Links and Authority 1542
Chapter 12 – Local and Map SEO 1680
Chapter 13 – Log-File and Server Analysis 1804
Chapter 14 – GEO, AEO and the End-to-End Technical Audit 1940



PART I

Beginner Practice

Apply the foundations to small, runnable projects. Drill the patterns that will appear in every program you ever write.

Chapter 1 – How Search Engines Work · Chapter 2 – robots.txt and Crawl Directives · Chapter 3 – XML Sitemaps and URL Discovery · Chapter 4 – HTTP Status Codes and Redirects

Chapter 1 · How Search Engines Work

Crawl, index and rank — what a bot sees, crawl budget, and how URLs are discovered.

Chapter Skill Map

What this chapter adds: In How Search Engines Work, the focus is crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered. You practise it across 30 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: This is the starting layer, so the main job is learning to connect visible code with visible output.

Where it leads: Chapter 2, robots.txt and Crawl Directives, reuses this chapter's discipline with a new kind of problem.

1 **Projects 1-10** Recognise the pattern and read the code slowly. *name it first*	2 **Projects 11-20** Trace similar programs with less prompting and explain what changed.	3 **Projects 21-30** Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

A search engine is three machines in a row: crawl → index → rank. *find it, understand it, order it*

Type a query and the answer feels instant, but the work happened long before you pressed Enter. A search engine runs three separate jobs. First it crawls — automated programs called *bots* follow links from page to page, fetching HTML. Then it indexes — it reads each page, pulls out the words and signals, and stores them in a giant lookup structure. Only then can it rank — when a query arrives, it pulls every candidate page from the index and orders them by how well they answer the question.

Technical SEO is the craft of making those three jobs easy. A page that the crawler never reaches cannot be indexed; a page that is indexed but mis-understood cannot rank; a page that ranks but loads slowly loses the click. So the whole field reduces to one question asked three ways: *can the bot reach this, can it understand this, and have I given it reasons to prefer this?*

The crawler's view of your site is brutally simple — it is a graph. Each URL is a node; each link is a directed edge. The bot starts at a seed (your homepage or a sitemap), follows edges, and keeps a `seen` set so it never visits the same page twice. That single data structure is the entire crawler:

```
from collections import deque

links = {'home': ['about', 'blog'], 'about': ['home'],
         'blog': ['post'], 'post': []}
seen, order, q = {'home'}, [], deque(['home'])
while q:
    page = q.popleft()
    order.append(page)
```

```
    for nxt in links[page]:
        if nxt not in seen:
            seen.add(nxt); q.append(nxt)
print(' -> '.join(order))
```

```
output
home -> about -> blog -> post
```

Read that and you have read Googlebot in miniature: a queue of URLs to fetch, a set of URLs already fetched, and a loop. Everything technical SEO does — robots.txt, sitemaps, internal linking, redirects — is just changing which edges exist and which nodes the bot is allowed to reach.

change the graph, change what gets found

Reaching pages is not free. A bot will only fetch so many URLs from your site in a given window — its `crawl budget`. Spend it on duplicate, broken, or junk URLs and your important pages wait days to be re-crawled. So a recurring move in this book is *pruning the graph*: block the junk, surface the priorities, and let the limited budget land where it matters.

CRAWLER (BOT)

An automated program that fetches pages and follows their links to discover more. Googlebot, Bingbot and others identify themselves in the `User-Agent` string of each request.

CRAWL

Phase one: discovering and fetching URLs by walking the link graph from a set of seed URLs and any submitted sitemaps.

INDEX

Phase two: parsing each fetched page into terms and signals and storing them so they can be looked up by query. A page not in the index can never appear in results.

RANK

Phase three: for a given query, ordering the indexed candidates by relevance and quality signals.

CRAWL BUDGET

The number of URLs a bot will fetch from one site in a given period. Wasting it on duplicates or errors delays the crawling of pages that matter.

INVERTED INDEX

The core index structure: a map from each term to the set of pages that contain it, so a query becomes a fast set lookup instead of a scan of every page.

The crawler sees a graph, not a website

A human sees menus, headlines and images. A crawler sees URLs and the links between them — a directed graph. The most important consequence: if no link points to a page, the crawler may never find it. Such *orphan pages* are invisible no matter how good their content. Finding them is pure graph reachability — start at the seed, collect everything reachable, and whatever is left over is orphaned:

```
all_pages = {'home', 'about', 'blog', 'secret'}
links = {'home': ['about', 'blog'], 'about': [], 'blog': []}
reachable, stack = set(), ['home']
while stack:
    p = stack.pop()
    if p in reachable:
        continue
    reachable.add(p)
    stack.extend(links.get(p, []))
print(sorted(all_pages - reachable))
```

```
output
['secret']
```

`secret` exists but nothing links to it, so the crawl never lands there. The fix is not better content — it is an *edge*: link to the page, or list it in a sitemap.

Discovered is not indexed; indexed is not ranked

These three sets shrink as you go: every URL the bot *discovers* is a superset of what it *indexes*, which is a superset of what actually *ranks* for any query. A page can drop out at each step — discovered but blocked, indexed but thin, indexed but out-ranked. Naming which set a page is stuck in is the first diagnostic move of any audit.

discovered ⊇ indexed ⊇ ranking

```
discovered = {'/', '/a', '/b', '/dup', '/blocked'}
indexed    = {'/', '/a', '/b'}          # /dup folded, /blocked disallowed
```

```
print('not indexed:', sorted(discovered - indexed))
```

```
output
not indexed: ['/blocked', '/dup']
```

Indexing: a page becomes a bag of terms

To make pages searchable the engine tokenizes each one — lowercases the text, splits on non-letters, and records which terms appear where. Build that for several pages and you have an inverted index: term → set of pages. A one-word query is then just a dictionary lookup, not a scan of the whole web.

```
import re
docs = {1: 'Fast SEO audit', 2: 'SEO crawl budget', 3: 'Audit your site'}
index = {}
for pid, text in docs.items():
    for term in re.findall(r'[a-z]+', text.lower()):
        index.setdefault(term, set()).add(pid)
print('audit ->', sorted(index['audit']))
print('seo   ->', sorted(index['seo']))
```

```
output
audit -> [1, 3]
seo   -> [1, 2]
```

Notice the engine never re-reads the documents to answer a query — it reads the *index*. That is why how your words are written (and whether the crawler can read them at all) decides everything downstream.

Ranking: order the candidates by a score

Once the index hands back the candidate pages, ranking sorts them by a score. Real scoring blends hundreds of signals; the oldest and most intuitive is term frequency — how often the query word appears, relative to page length. Here three pages compete for the word "seo":

```
pages = {'A': 'seo seo audit', 'B': 'seo crawl index rank',
         'C': 'marketing tips'}
def score(text, term):
    words = text.split()
    return round(words.count(term) / len(words), 3)
ranked = sorted(pages, key=lambda p: score(pages[p], 'seo'), reverse=True)
print(ranked)
```

```
output
['A', 'B', 'C']
```

Page A wins on raw frequency, B is on-topic but diluted, C does not mention the term at all. Modern engines add quality, links and intent on top — but the shape is always the same: *retrieve candidates from the index, score each,*

sort descending. retrieve → score → sort

Example 1

A site has pages home, services, blog, and a checkout page reachable only from services. The homepage links to services and blog. Is checkout crawlable?

SOLUTION

1. Build the reachable set from the seed `home`: home links to `services` and `blog`, so both are reached.
2. From services we reach `checkout` — there is an edge services → checkout, so the bot follows it.
3. Every page is reachable from home, so checkout is crawlable even though only one page links to it.

Yes — checkout is crawlable because a path home → services → checkout exists.

reachability, not link count, is what matters

Example 2

Your crawl budget is 3 fetches this visit. The frontier holds [home (depth 0), a (1), b (1), c (2)]. Which pages get crawled, and what is the risk if a, b are duplicates?

SOLUTION

1. A bot generally crawls breadth-first, so it spends the budget on the shallowest URLs first: `home, a, b`.
2. Page c (depth 2) is left for a later visit — the budget ran out before reaching it.
3. If a and b are duplicates, two of three fetches were wasted on the same content, and the unique page c was the one delayed.

home, a, b are crawled; c waits. Duplicates burned the budget and starved a real page — the core argument for pruning duplicate URLs. budget spent on junk is budget stolen from value

Example 3

For the query "seo", the index returns pages A ("seo seo seo", 3 words) and B ("seo guide for beginners", 4 words). Rank them by term frequency.

SOLUTION

1. Page A: the term "seo" appears 3 times in 3 words → frequency 3/3 = `1.0`.
2. Page B: "seo" appears 1 time in 4 words → frequency 1/4 = `0.25`.
3. Sort descending by score: A (1.0) before B (0.25).

Ranked A, then B. Raw term frequency favours A — though a real engine would notice A is keyword-stuffed and discount it. frequency is a start, not the finish

⚡ Mental-Math Shortcut

Reach, read, reason. Before any tactic, ask the three questions in order: can the bot *reach* the page (crawl), can it *read* the meaning (index), have I given it *reasons* to prefer it (rank)? A problem almost always lives in the earliest phase you have not yet checked — fix crawl before you obsess over ranking.

⚠ Common Mistakes

- Optimising ranking signals on a page the crawler cannot reach — no internal link or sitemap entry means it is never even a candidate.
- Assuming "indexed" means "ranking". A page can sit in the index and still never surface because stronger pages out-score it for every query.
- Letting duplicate and parameter URLs soak up crawl budget, so important pages are re-crawled days late.
- Confusing what humans see with what the bot sees — JavaScript-built links and menus may not be in the HTML the crawler first fetches.

This chapter contains 30 hands-on mini-projects (Projects 1–30). Type each one out, run it, and experiment.

Project 1 Model a Tiny Web as a Link Graph

💡 The Idea Behind It

Everything a search engine does begins with one data structure: a directed graph where pages are keys and the lists they map to are the links leaving them. This project encodes that graph as a Python dict, then derives two headline numbers, node count and edge count, by measuring the dict's length and summing the lengths of its values. It proves the web is just nodes and arrows you can count.

What You'll Build & Learn

Represent a five-page site as an adjacency dict and report how many pages and links it contains.

- Pages are nodes, links are directed edges
- A dict of lists is an adjacency list
- Summing value lengths counts total edges

🧭 Skill Roadmap

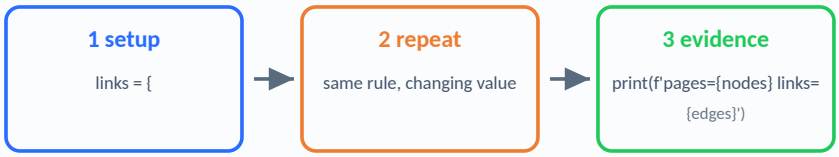
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 2 changes the surface problem while preserving the underlying method. *progress check*

Reading target: model a tiny web as a link graph is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'pages={nodes} links={edges}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`edges = sum(len(v) for v in links.values())`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'pages={nodes} links={edges}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `edges = sum(len(v) for v in links.values())`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Model a Tiny Web as a Link Graph: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: links, home, about, blog, post1. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: links = { | Core: edges = sum(len(v) for v in links.values()) | Evidence: print(f'pages={nodes} links={edges}')

🛠️ Reader to Writer Bridge

1. Read: explain model a tiny web as a link graph in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 2 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'pages={nodes} links={edges}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of model a tiny web as a link graph and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A dict of `url -> [linked urls]` is an adjacency list, the standard graph representation; the number of pages is the key count and the number of links is the total length of all value lists.

```
links = {
    'home': ['about', 'blog'],
    'about': ['home'],
    'blog': ['post1', 'post2'],
    'post1': [],
    'post2': ['blog'],
}
nodes = len(links)
edges = sum(len(v) for v in links.values())
print(f'pages={nodes} links={edges}')
```

```
output
pages=5 links=6
```

How it works:

1. The links dict maps each page name to a list of the pages it points to, so home links out to about and blog.
2. nodes = len(links) counts the dict keys, giving five pages.
3. edges sums len(v) across every value list, totalling six outgoing links.
4. The f-string prints both counts as pages=5 links=6.

Try it:

1. Add a page that links to two existing pages and re-run the counts.
2. Compute the average out-degree by dividing edges by nodes.
3. Find pages with zero outgoing links by filtering for empty value lists.
4. Add a page `'contact': []` and watch the page count rise but edge count stay.

⚠ Common Mistakes

- Counting `len(links)` as edges instead of nodes confuses pages with links.
- Summing `len(v)` over a dict of dicts (not lists) would count keys, not links.
- A page that links to itself still counts as one edge, which is easy to overlook.

Skills gained adjacency list dict comprehension graph modelling

→ Where You'll Use This: Every crawl audit, internal-link map, and PageRank calculation starts from exactly this adjacency representation of a site's link structure.

⚡ Mental Model

Think of the dict like an airport departures board: each city lists the destinations you can fly to directly from there.

Project 2 Crawl a Tiny Web with BFS

💡 The Idea Behind It

A crawler discovers pages by following links outward from a seed, and breadth-first search captures the natural order: visit the seed, then everything one click away, then two clicks away. This code uses a deque as a FIFO queue plus a seen set to avoid revisiting, popping from the left so the nearest pages always come first. The output traces that expanding wavefront across the link graph.

What You'll Build & Learn

Crawl the sample site breadth-first from home and print the exact order pages are visited.

- BFS visits nearest pages first
- A deque gives FIFO queue behaviour
- A seen set prevents revisiting pages

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 1: model a tiny web as a link graph. Project 3 changes the surface problem while preserving the underlying method.

progress check

Reading target: crawl a tiny web with bfs is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print(' -> '.join(order))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if nxt not in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' -> '.join(order))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt not in seen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Crawl a Tiny Web with BFS: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, about. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if nxt not in seen: | Evidence: print(' -> '.join(order))

🛠️ Reader to Writer Bridge

1. Read: explain crawl a tiny web with bfs in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 1.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 3 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' -> '.join(order))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of crawl a tiny web with bfs and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A queue (FIFO) plus a `seen` set is the whole crawler: dequeue a page, record it, enqueue its unseen links. The `seen` set stops the crawler revisiting pages and looping forever.

```
from collections import deque

links = {
    'home': ['about', 'blog'],
    'about': ['home'],
    'blog': ['post1', 'post2'],
    'post1': [],
    'post2': ['blog'],
}
seen, order, q = {'home'}, [], deque(['home'])
while q:
    page = q.popleft()
    order.append(page)
    for nxt in links[page]:
        if nxt not in seen:
            seen.add(nxt)
            q.append(nxt)
print(' -> '.join(order))
```

```
output
home -> about -> blog -> post1 -> post2
```

How it works:

1. seen, order, and q start with home as the seed in all three.
2. While the queue is non-empty, popleft removes the front page and appends it to order.
3. Each linked page not already in seen is added to seen and enqueued.
4. Joining order with arrows shows home reaching about and blog before their children.

Try it:

1. Add a fresh page linked from post1 and confirm it appears last.
2. Track and print each page's discovery distance from the seed.
3. Seed the crawl from blog instead and compare the visit order.
4. Swap the deque for a stack (`pop()`) to crawl depth-first and compare the order.

⚠ Common Mistakes

- Adding to `seen` only when you dequeue, not when you enqueue, lets the same URL enter the queue twice.
- Using a list with `.pop(0)` instead of a deque works but is O(n) per pop on large crawls.
- Forgetting a page key in the dict raises KeyError mid-crawl.

Skills gained BFS traversal | deque queue | set deduplication

→ Where You'll Use This: Real crawlers like Googlebot use breadth-first ordering to spread limited crawl budget evenly across a site rather than tunnelling into one branch.

⚡ Mental Model

BFS is ripples spreading from a stone dropped in water: every ring is fully explored before the next one forms.

Project 3 Crawl Depth-First and Compare the Order

💡 The Idea Behind It

Swap the breadth-first queue for a stack and the identical graph is explored in a completely different shape: the crawler plunges down one branch to its end before backing up to try alternatives. Here a list used with pop() and append() acts as the LIFO stack, and pushing children in reversed order keeps left-to-right priority. The seen check guards against revisiting, mirroring depth-first traversal.

What You'll Build & Learn

Crawl the same site depth-first using a stack and contrast the resulting order with BFS.

- A stack drives depth-first exploration
- pop() removes the most recently added item
- reversed() preserves natural child ordering

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 2: crawl a tiny web with bfs. Project 4 changes the surface problem while preserving the underlying method. *progress check*

Reading target: crawl depth-first and compare the order is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print(' -> '.join(order))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if page in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' -> '.join(order))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if page in seen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Crawl Depth-First and Compare the Order: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: links, home, about, blog, post1. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: links = { | Core: if page in seen: | Evidence: print(' -> '.join(order))

🛠️ Reader to Writer Bridge

1. Read: explain crawl depth-first and compare the order in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 2.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 4 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' -> '.join(order))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of crawl depth-first and compare the order and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Popping from the end of a list (LIFO stack) drives depth-first traversal; reversing the link list before pushing keeps the visit order left-to-right so the output is intuitive.

```
links = {
    'home': ['about', 'blog'],
    'about': ['home'],
    'blog': ['post1', 'post2'],
    'post1': [],
    'post2': ['blog'],
}
seen, order, stack = set(), [], ['home']
while stack:
    page = stack.pop()
    if page in seen:
        continue
    seen.add(page)
    order.append(page)
    for nxt in reversed(links[page]):
        if nxt not in seen:
            stack.append(nxt)
print(' -> '.join(order))
```

```
output
home -> about -> blog -> post1 -> post2
```

How it works:

1. stack starts holding only home, with empty seen and order.
2. pop() takes the top page; if already seen it is skipped via continue.
3. An unseen page is recorded in seen and order, then its children push on in reversed order.
4. The arrows show the crawler diving into one branch before exploring siblings.

Try it:

1. Remove the reversed() call and observe how the order flips.
2. Add a deeper chain of three pages and trace the dive.
3. Convert this into a recursive depth-first function instead of a stack.
4. Remove `reversed(...)` and observe the reordered output.

⚠ Common Mistakes

- Without `reversed(...)` the children are visited right-to-left, which surprises readers comparing to BFS.
- Checking `seen` only at enqueue time (not at pop) can still record a page twice because a node may be pushed before it is first visited.
- Using `stack.pop(0)` turns the stack back into a queue and silently gives BFS order.

Skills gained DFS traversal | stack with list | reversed iteration

→ **Where You'll Use This:** Understanding DFS versus BFS explains why crawler architecture matters: depth-first can strand a wide site's shallow pages while it tunnels into deep paths.

⚡ Mental Model

DFS is exploring a maze by always taking the first unexplored corridor and only backtracking when you hit a dead end.

Project 4 Crawl Within a Depth Limit

💡 The Idea Behind It

Crawlers cap how many clicks deep they descend so endless pagination and faceted navigation never drain their budget. This BFS carries each page's depth alongside it in the queue tuple, and when a page sits at the depth limit the loop hits continue and refuses to enqueue its children. With limit one, only home and its direct neighbours survive, leaving deeper pages unreached and unfetched.

What You'll Build & Learn

Run a depth-limited BFS so only pages within one hop of the seed are reached, then list them.

- Depth limits bound crawl reach
- Queue entries can carry their own depth
- continue skips enqueuing past the limit

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 3: crawl depth-first and compare the order. Project 5 changes the surface problem while preserving the underlying method.

progress check

Reading target: crawl within a depth limit is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print(sorted(reached))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if d == limit:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(sorted(reached))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if d == limit:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Crawl Within a Depth Limit: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, about. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if d == limit: | Evidence: print(sorted(reached))

🛠️ Reader to Writer Bridge

1. Read: explain crawl within a depth limit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 3.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 5 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(sorted(reached))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of crawl within a depth limit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Pairing each queued page with its depth and refusing to expand nodes at the limit confines the crawl to a radius-N ball around the seed.

```
from collections import deque

links = {
    'home': ['about', 'blog'],
    'about': ['team'],
    'blog': ['post1'],
    'post1': ['deep'],
    'team': [], 'deep': [],
}
limit = 1
reached, q = {'home': 0}, deque([('home', 0)])
while q:
    page, d = q.popleft()
    if d == limit:
        continue
    for nxt in links[page]:
        if nxt not in reached:
            reached[nxt] = d + 1
            q.append((nxt, d + 1))
print(sorted(reached))
```

```
output
['about', 'blog', 'home']
```

How it works:

1. reached maps home to depth zero and the queue holds the (home, 0) tuple.
2. Each iteration unpacks page and its depth d from the front of the queue.
3. If d equals the limit, continue stops the page's children being added.
4. Otherwise unseen children get depth d+1 and are enqueued, yielding about, blog, home.

Try it:

1. Raise the limit to two and see which deeper pages appear.
2. Print each reached page alongside its recorded depth.
3. Change the seed and observe how the reachable set shifts.
4. Raise `limit` to 2 and watch `post1`'s neighbour `deep` appear.

⚠ Common Mistakes

- Comparing `d > limit` instead of `d == limit` before skipping expands one level too far.
- Tracking depth in a separate structure that drifts out of sync with the queue.
- Re-adding an already-reached page at a greater depth and overwriting its smaller depth.

Skills gained `depth-limited BFS` `tuple unpacking` `frontier pruning`

→ **Where You'll Use This:** Click-depth limits map directly to real crawl-budget policy: pages buried many clicks from the homepage are crawled rarely or skipped entirely.

⚡ Mental Model

It is like exploring a city but refusing to walk more than one block from the hotel, no matter how many streets branch off.

Project 5 Watch the Crawl Frontier Evolve

💡 The Idea Behind It

The frontier is the live queue of URLs a crawler still has to fetch, and printing it at every step turns an abstract algorithm into a visible wavefront. This BFS prints list(q) before each pop and once more after the loop, so you watch the queue swell as children arrive and shrink as pages are processed. The snapshots reveal exactly how the breadth-first sweep moves through the graph.

What You'll Build & Learn

Instrument a BFS to print the queue at each step, making the crawl frontier visible as it evolves.

- The frontier is the pending-fetch queue
- Printing state exposes algorithm behaviour
- The frontier grows then drains to empty

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 4: crawl within a depth limit. Project 6 changes the surface problem while preserving the underlying method. *progress check*

Reading target: watch the crawl frontier evolve is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print('frontier:', list(q))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if nxt not in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('frontier:', list(q))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt not in seen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Watch the Crawl Frontier Evolve: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, seen. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if nxt not in seen: | Evidence: print('frontier:', list(q))

🛠️ Reader to Writer Bridge

1. Read: explain watch the crawl frontier evolve in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 4.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 6 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('frontier:', list(q))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of watch the crawl frontier evolve and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Printing the queue before each pop shows the frontier expanding as new links are discovered and shrinking as pages are processed, ending empty when the crawl is done.

```python
from collections import deque

links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c', 'd'],
    'c': [], 'd': [],
}
seen, q = {'home'}, deque(['home'])
while q:
    print('frontier:', list(q))
    page = q.popleft()
    for nxt in links[page]:
        if nxt not in seen:
            seen.add(nxt)
            q.append(nxt)
print('frontier:', list(q))
```

```
output
frontier: ['home']
frontier: ['a', 'b']
frontier: ['b', 'c']
frontier: ['c', 'd']
frontier: ['d']
frontier: []
```

How it works:

1. seen and the queue both begin with home only.
2. Each loop prints the current frontier as a list before popping the front page.
3. Unseen children are added to seen and the queue, so the printed frontier shifts.
4. A final print after the loop shows the empty frontier once every page is fetched.

Try it:

1. Also print the seen set at each step beside the frontier.
2. Add a wider node with three children and watch the surge.
3. Print the size of the frontier instead of its contents.
4. Add an extra page linked from `d` and watch the frontier refill late.

⚠ Common Mistakes

- Printing after the pop instead of before hides the page about to be processed.
- Mutating `q` while iterating its snapshot would corrupt the displayed frontier.
- Forgetting the final `print` after the loop omits the empty-frontier end state.

Skills gained `state logging` `frontier inspection` `BFS instrumentation`

→ **Where You'll Use This:** Logging the frontier is how engineers debug a stalled or runaway crawler in production, confirming which URLs are queued versus already processed.

⚡ Mental Model

The frontier is a to-do list of doors not yet opened: you add new doors as you find them and cross off each one you walk through.

Project 6 Separate Discovered from Indexed Pages

💡 The Idea Behind It

Discovering a URL and actually crawling it are two distinct events, and a limited budget splits them apart. This code keeps a discovered set and a visited list, halting the BFS once visited reaches the budget of two even though more URLs are queued. Subtracting visited from discovered yields the backlog: pages the crawler knows about but has not yet fetched this pass.

What You'll Build & Learn

Crawl under a budget of two fetches, then separate the indexed pages from the backlog still queued.

- Discovery and crawling are separate stages
- A budget halts the loop early
- Set difference reveals the backlog

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 5: watch the crawl frontier evolve. Project 7 changes the surface problem while preserving the underlying method.

progress check

Reading target: separate discovered from indexed pages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print('still queued:', sorted(backlog))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if nxt not in discovered:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('still queued:', sorted(backlog))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt not in discovered:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Separate Discovered from Indexed Pages: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, budget. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if nxt not in discovered: | Evidence: print('still queued:', sorted(backlog))

🛠️ Reader to Writer Bridge

1. Read: explain separate discovered from indexed pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 5.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 7 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('still queued:', sorted(backlog))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of separate discovered from indexed pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

The `discovered` set grows whenever a link is found, but `visited` only grows when a page is actually fetched; with a budget of 2 the difference is the unfetched backlog.

```
from collections import deque

links = {
    'home': ['a', 'b', 'c'],
    'a': ['d'], 'b': ['e'], 'c': [],
    'd': [], 'e': [],
}
budget = 2
discovered, visited, q = {'home'}, [], deque(['home'])
while q and len(visited) < budget:
    page = q.popleft()
    visited.append(page)
    for nxt in links[page]:
        if nxt not in discovered:
            discovered.add(nxt)
            q.append(nxt)
backlog = discovered - set(visited)
print('indexed:', visited)
print('still queued:', sorted(backlog))
```

```
output
indexed: ['home', 'a']
still queued: ['b', 'c', 'd']
```

How it works:

1. discovered holds home, visited is empty, and the queue starts at home.
2. The loop runs while the queue has items and visited is below the budget of two.
3. Each popped page is appended to visited and its unseen children added to discovered.
4. backlog = discovered minus visited lists b, c, d as still queued.

Try it:

1. Raise the budget to four and see the backlog shrink.
2. Print the discovered set's full size next to visited.
3. Track the order in which URLs entered the discovered set.
4. Raise `budget` to 5 and confirm the backlog empties.

⚠ Common Mistakes

- Adding the seed to `visited` up front double-counts it against the budget.
- Comparing `discovered` to `visited` as lists (not sets) breaks the set subtraction.
- Stopping on `len(q)` instead of `len(visited)` miscounts the budget.

Skills gained `set difference` `budget gating` `backlog tracking`

→ **Where You'll Use This:** This mirrors the gap in Search Console between Discovered currently not indexed and actually indexed pages on a large site.

⚡ Mental Model

Discovery is adding books to a reading list; the budget is how many you actually finish before the library closes.

Project 7 Find Orphan Pages No Link Reaches

💡 The Idea Behind It

An orphan page exists on a site yet no internal link points to it, so a link-following crawler never reaches it. This code crawls breadth-first from home to build the seen set of everything reachable, then subtracts that from the full set of pages in the links dict. Whatever remains, here secret and old, is orphaned despite secret itself linking outward.

What You'll Build & Learn

Crawl from the seed, then compute which pages exist but are never reached by any link.

- Orphan pages have no inbound links
- Reachability comes from a crawl traversal
- set(links) minus seen finds orphans

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 6: separate discovered from indexed pages. Project 8 changes the surface problem while preserving the underlying method. *progress check*

Reading target: find orphan pages no link reaches is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print('orphans:', sorted(orphans))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if nxt not in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('orphans:', sorted(orphans))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt not in seen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Orphan Pages No Link Reaches: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, about. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if nxt not in seen: | Evidence: print('orphans:', sorted(orphans))

🛠️ Reader to Writer Bridge

1. Read: explain find orphan pages no link reaches in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 6.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 8 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('orphans:', sorted(orphans))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find orphan pages no link reaches and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Crawl from the seed to build the `seen` set, then subtract it from all known page keys; whatever is left was never reachable by following links.

```python
from collections import deque

links = {
    'home': ['about', 'blog'],
    'about': [], 'blog': ['post1'],
    'post1': [],
    'secret': ['home'],
    'old': [],
}
seen, q = {'home'}, deque(['home'])
while q:
    page = q.popleft()
    for nxt in links[page]:
        if nxt not in seen:
            seen.add(nxt)
            q.append(nxt)
orphans = set(links) - seen
print('orphans:', sorted(orphans))
```

```
output
orphans: ['old', 'secret']
```

How it works:

1. A standard BFS from home fills the seen set with every reachable page.
2. set(links) is the full inventory of pages declared in the graph.
3. Subtracting seen from that inventory leaves the unreached pages.
4. Sorting gives orphans as old and secret, even though secret links to home.

Try it:

1. Add an internal link to old and confirm it stops being an orphan.
2. List orphans together with any links they point out.
3. Seed the crawl from a sitemap set instead of home.
4. Add a link from `home` to `old` and watch `old` stop being an orphan.

⚠ Common Mistakes

- Listing pages with no outbound links as orphans confuses out-links with in-links.
- Forgetting that a page reachable only via an orphan is itself an orphan.
- Treating the seed as a possible orphan even though it is always reachable.

Skills gained reachability analysis set difference orphan detection

→ Where You'll Use This: Finding orphans is a staple SEO audit: pages with no internal links rarely get indexed and leak no link equity to the rest of the site.

Mental Model

An orphan page is a room in a house with no door: it is fully built but nobody can ever walk in.

Project 8 Normalize and Dedupe URLs

The Idea Behind It

One page can hide behind many URLs that differ only in case, a trailing slash, or a fragment, so a crawler folds them to a single canonical form before fetching. The normalize function splits scheme from host and path, drops anything after a hash, strips a trailing slash, and lowercases the host. Feeding the results through a set comprehension collapses the four raw URLs into two unique ones.

What You'll Build & Learn

Normalize messy URLs and deduplicate them into the canonical set a crawler would actually fetch.

- URLs vary without changing the page
- Hosts are case-insensitive, paths are not
- Fragments and trailing slashes are noise

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 7: find orphan pages no link reaches. Project 9 changes the surface problem while preserving the underlying method.

progress check

Reading target: normalize and dedupe urls is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `raw = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print('\n'.join(unique))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`raw = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if '/' in rest:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('\n'.join(unique))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if '/' in rest:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Normalize and Dedupe URLs: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: raw, https, Site, com, page. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: raw = [| Core: if '/' in rest: | Evidence: print('\n'.join(unique))

🛠️ Reader to Writer Bridge

1. Read: explain normalize and dedupe urls in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 7.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 9 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('\n'.join(unique))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `raw = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of normalize and dedupe urls and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Lowercasing the host, dropping the `#fragment`, and trimming a trailing slash collapses cosmetic variants; feeding the results into a set removes the now-identical duplicates.

```
raw = [
    'https://Site.com/page/',
    'https://site.com/page',
    'https://SITE.com/page#section',
    'https://site.com/other/',
]

def normalize(u):
    scheme, rest = u.split('://', 1)
    if '/' in rest:
        host, path = rest.split('/', 1)
        path = '/' + path
    else:
        host, path = rest, ''
    path = path.split('#', 1)[0]
    if len(path) > 1 and path.endswith('/'):
        path = path[:-1]
    return scheme + '://' + host.lower() + path

unique = sorted({normalize(u) for u in raw})
print('\n'.join(unique))
```

```
output
https://site.com/other
https://site.com/page
```

How it works:

1. normalize splits off the scheme, then separates host from path.
2. It cuts everything after a hash to discard the fragment.
3. A trailing slash on a non-root path is removed and the host is lowercased.
4. A set comprehension over raw drops duplicates, leaving two sorted canonical URLs.

Try it:

1. Also strip a leading www. from the host before comparing.
2. Sort and drop tracking query parameters like utm_source.
3. Add an http versus https pair and decide how to fold them.
4. Add `'https://site.com/page?x=1'` and decide whether the query should survive.

⚠ Common Mistakes

- Lowercasing the whole URL breaks case-sensitive paths; only the host is safe to lowercase.
- Stripping a trailing slash from the root path `'/'` would wrongly empty it.
- Dropping the query string along with the fragment loses meaningful parameters.

Skills gained URL normalization string parsing set deduplication

→ Where You'll Use This: URL normalization underpins canonicalization audits and stops crawlers wasting budget indexing the same page under dozens of equivalent addresses.

Mental Model

Normalization is like recognising that Bob, bob, and Bob. all name the same person before you add them to a guest list.

Project 9 Resolve Relative Links Against a Base

The Idea Behind It

Pages link with relative paths like boots or /sale, but a crawler can only fetch absolute URLs, so resolving a relative link against the current page is a core step. The join function splits the base into scheme, host, and path, then branches: a leading slash means root-relative, otherwise the link is appended to the base's parent directory. Each relative form resolves to a full, fetchable address.

What You'll Build & Learn

Resolve a set of relative links against a base URL to produce absolute, crawlable URLs.

- Crawlers need absolute URLs to fetch
- A leading slash means root-relative
- Otherwise resolve against the parent path

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 8: normalize and dedupe urls. Project 10 changes the surface problem while preserving the underlying method.

progress check

Reading target: resolve relative links against a base is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `base = 'https://shop.example/cat/shoes'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print(join(base, r))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`base = 'https://shop.example/cat/shoes'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if rel.startswith('/'):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(join(base, r))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if rel.startswith('/'):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Resolve Relative Links Against a Base: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: base, https, shop, example, cat. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: base = 'https://shop.example/cat/shoes' | Core: if rel.startswith('/'): | Evidence: print(join(base, r))

🛠️ Reader to Writer Bridge

1. Read: explain resolve relative links against a base in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 8.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 10 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(join(base, r))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `base = 'https://shop.example/cat/shoes'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of resolve relative links against a base and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

An absolute path (leading `/`) replaces the whole path, while a relative one is appended to the current path's parent directory; the scheme and host are carried over unchanged.

```
base = 'https://shop.example/cat/shoes'
rels = ['boots', '/sale', 'shoes?p=2']

def join(base, rel):
    scheme, rest = base.split('://', 1)
    host, _, path = rest.partition('/')
```

```
    path = '/' + path
    if rel.startswith('/'):
        new = rel
    else:
        parent = path.rsplit('/', 1)[0]
        new = parent + '/' + rel
    return scheme + '://' + host + new

for r in rels:
    print(join(base, r))
```

```
output
https://shop.example/cat/boots
https://shop.example/sale
https://shop.example/cat/shoes?p=2
```

How it works:

1. join splits the base into scheme, host, and the path after the host.
2. If the relative link starts with a slash it replaces the path entirely.
3. Otherwise rsplit drops the base's last segment and appends the relative link.
4. The loop prints boots, /sale, and shoes?p=2 each resolved to a full URL.

Try it:

1. Handle a ../ parent reference that climbs one directory.
2. Support a protocol-relative link beginning with double slashes.
3. Resolve a bare fragment like #top to the base page.
4. Add a `../` relative link and decide how to climb a directory.

⚠ Common Mistakes

- Joining `boots` to the full path instead of its parent yields `/cat/shoes/boots`, not `/cat/boots`.
- Treating a `/sale` absolute path like a relative one doubles up the directory.
- Splitting the host off with `split('/')` instead of `partition('/')` raises if there is no path.

Skills gained URL resolution | string partitioning | path joining

→ Where You'll Use This: Every HTML parser in a crawler must resolve relative hrefs to absolute URLs before queuing them, exactly as the browser address bar does.

Mental Model

Resolving relative links is reading directions from where you stand: turn left means something different on every street corner.

Project 10 Spend a Crawl Budget of K Fetches

The Idea Behind It

Googlebot never fetches every URL on every visit; it spends a finite crawl budget, and breadth-first order decides which pages win this pass. This code crawls with a deque but stops once the crawled list reaches the budget of four, leaving deeper pages untouched. The output reports four of seven pages fetched, showing budget directly capping coverage regardless of how many URLs exist.

What You'll Build & Learn

Crawl breadth-first until a budget of four fetches is spent, then report coverage against the total.

- Crawl budget caps fetches per pass
- BFS order picks the lucky pages
- Coverage is crawled over total

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 9: resolve relative links against a base. Project 11 changes the surface problem while preserving the underlying method.

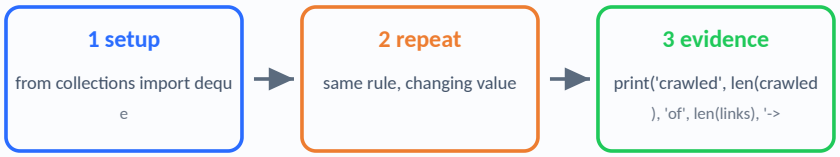
progress check

Reading target: spend a crawl budget of k fetches is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print('crawled', len(crawled), 'of', len(links), '->', crawled)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if nxt not in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('crawled', len(crawled), 'of', len(links), '->', crawled)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt not in seen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Spend a Crawl Budget of K Fetches: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, budget. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if nxt not in seen: | Evidence: print('crawled', len(crawled), 'of', len(links), '->', crawled)

🛠️ Reader to Writer Bridge

1. Read: explain spend a crawl budget of k fetches in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 9.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 11 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('crawled', len(crawled), 'of', len(links), '->', crawled)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of spend a crawl budget of k fetches and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Stopping the BFS once `len(crawled)` hits the budget means the shallowest, earliest-discovered pages win the limited fetches, which mirrors how crawl budget favours well-linked pages.

```
from collections import deque

links = {
    'home': ['a', 'b', 'c'],
    'a': ['d'], 'b': ['e'], 'c': ['f'],
    'd': [], 'e': [], 'f': [],
```

```
}
budget = 4
crawled, seen, q = [], {'home'}, deque(['home'])
while q and len(crawled) < budget:
    page = q.popleft()
    crawled.append(page)
    for nxt in links[page]:
        if nxt not in seen:
            seen.add(nxt)
            q.append(nxt)
print('crawled', len(crawled), 'of', len(links), '->', crawled)
```

```
output
crawled 4 of 7 -> ['home', 'a', 'b', 'c']
```

How it works:

1. crawled, seen, and the queue initialise with home as the seed.
2. The loop continues while the queue holds pages and crawled is below the budget.
3. Each popped page is appended and its unseen children enqueued for later.
4. The print reports crawled 4 of 7, naming home, a, b, c.

Try it:

1. Compute and print the percentage of pages crawled.
2. Raise the budget until full coverage is reached.
3. Switch to a stack and see which pages the budget catches.
4. Raise the budget to 7 and confirm the whole site is crawled.

⚠ Common Mistakes

- Checking the budget after appending lets the crawl overshoot by one.
- Counting discovered links against the budget instead of actual fetches.
- Using a stack so the budget is spent diving one branch instead of spreading wide.

Skills gained budget capping | BFS traversal | coverage reporting

→ Where You'll Use This: Crawl-budget thinking guides large-site SEO: you prioritise high-value URLs near the homepage because the engine will not reach everything each visit.

Mental Model

Crawl budget is a shopping trip with a fixed cart: you grab the nearest aisles first and leave the back of the store for next time.

Project 11 Count Inbound Links per Page

The Idea Behind It

In-degree, the count of pages linking to a given URL, is the simplest popularity signal and the seed idea behind PageRank: more inbound links usually means a more important page. This code walks every target list in the graph and increments a Counter for each link's destination. Iterating the sorted pages then prints each one's inbound tally, turning raw links into a ranking input.

What You'll Build & Learn

Count how many internal links point to each page and print the in-degree per URL.

- In-degree counts inbound links
- It is a basic popularity signal
- Counter tallies destinations cleanly

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 10: spend a crawl budget of k fetches. Project 12 changes the surface problem while preserving the underlying method.

progress check

Reading target: count inbound links per page is the surface task; the transferable Python move is to mark setup, rule, and output before running.

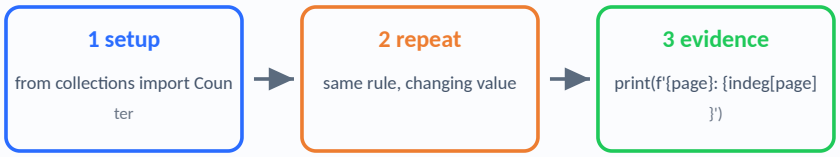
setup -> rule -> result

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{page}: {indeg[page]}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`for targets in links.values():`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{page}: {indeg[page]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for targets in links.values():`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Inbound Links per Page: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, Counter, links, home, about. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: for targets in links.values(): | Evidence: print(f'{page}: {indeg[page]}')

🛠️ Reader to Writer Bridge

1. Read: explain count inbound links per page in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 10.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 12 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{page}: {indeg[page]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count inbound links per page and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Iterating every link target and incrementing a `Counter` tallies in-degree; sorting the page keys before printing keeps the output deterministic.

```
from collections import Counter

links = {
    'home': ['about', 'blog'],
    'about': ['home'],
    'blog': ['home', 'post1'],
    'post1': ['blog'],
```

```
}
indeg = Counter()
for targets in links.values():
    for t in targets:
        indeg[t] += 1
for page in sorted(links):
    print(f'{page}: {indeg[page]}')
```

```
output
about: 1
blog: 2
home: 2
post1: 1
```

How it works:

1. An empty Counter named indeg will accumulate inbound counts.
2. The outer loop iterates the target lists, the inner loop each linked page.
3. indeg[t] += 1 credits one inbound link to every destination t.
4. Iterating sorted pages prints each name with its inbound count.

Try it:

1. Also compute and print each page's out-degree.
2. Find pages with zero inbound links automatically.
3. Sort the output by in-degree instead of by name.
4. Print out-degree alongside in-degree for each page.

⚠ Common Mistakes

- Counting `len(links[page])` measures out-degree, not in-degree.
- Skipping pages with zero inbound links because they never appear as a target.
- Iterating `links.keys()` for targets instead of the value lists.

Skills gained in-degree counting | Counter accumulation | nested iteration

→ Where You'll Use This: Internal-link counts feed link-equity audits, helping you confirm that money pages receive more internal links than peripheral ones.

Mental Model

In-degree is counting how many friends mention your name: the more shout-outs, the more the room assumes you matter.

Project 12 Find the Most-Linked Page

The Idea Behind It

If in-degree measures popularity, the page with the highest in-degree is the site's link hub, the spot where internal authority pools. This code first tallies inbound links into a Counter, then uses max over the sorted keys with a key function reading the count, so ties break deterministically by name. The winner is surfaced as the most-linked page along with its inbound total.

What You'll Build & Learn

Tally inbound links and surface the single most-linked page along with its in-degree.

- The link hub has the highest in-degree
- max with a key selects the winner
- Sorting first makes ties deterministic

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 11: count inbound links per page. Project 13 changes the surface problem while preserving the underlying method.

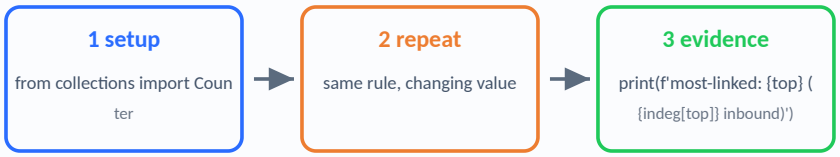
progress check

Reading target: find the most-linked page is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'most-linked: {top} ({indeg[top]} inbound)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`for targets in links.values():`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'most-linked: {top} ({indeg[top]} inbound)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for targets in links.values():`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find the Most-Linked Page: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, Counter, links, home, about. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: for targets in links.values(): | Evidence: print(f'most-linked: {top} ({indeg[top]} inbound)')

🛠️ Reader to Writer Bridge

1. Read: explain find the most-linked page in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 11.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 13 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'most-linked: {top} ({indeg[top]} inbound)')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find the most-linked page and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

After tallying in-degree, `max` over the sorted keys with an in-degree key function returns the most-linked page, with sorting making ties break alphabetically and deterministically.

```
from collections import Counter

links = {
    'home': ['about', 'blog'],
    'about': ['home', 'blog'],
    'blog': ['home'],
```

```
    'post1': ['blog'],
}
indeg = Counter()
for targets in links.values():
    for t in targets:
        indeg[t] += 1
top = max(sorted(indeg), key=lambda p: indeg[p])
print(f'most-linked: {top} ({indeg[top]} inbound)')
```

```
output
most-linked: blog (3 inbound)
```

How it works:

1. The Counter loop credits each destination an inbound link, as before.
2. sorted(indeg) orders candidate pages so equal counts break alphabetically.
3. max with key=lambda reading indeg[p] picks the highest-scoring page.
4. The f-string reports blog with three inbound links as the hub.

Try it:

1. Print the top three most-linked pages instead of one.
2. Break ties by out-degree when in-degrees are equal.
3. Report the least-linked page as the weakest spot.
4. Print the top two pages by slicing a sorted list instead of one `max`.

⚠ Common Mistakes

- Calling `max` over the unsorted dict makes ties resolve unpredictably.
- Passing `indeg.values()` to `max` loses which page the count belongs to.
- Forgetting pages with zero in-degree are absent from the Counter entirely.

Skills gained max with key | deterministic tie-break | Counter usage

→ Where You'll Use This: Spotting the internal link hub tells you where authority concentrates and whether it lands on a page you actually want to rank.

Mental Model

The most-linked page is the town square: every street eventually points there, so it is where the crowd gathers.

Project 13 Tokenize an HTML Body into Terms

The Idea Behind It

Before an engine can index a page it must strip the HTML and reduce the visible text to lowercase word tokens, the bridge from raw markup to indexable terms. This code uses one regular expression to replace every tag with a space and another to pull out runs of lowercase letters, then a Counter tallies them. The three most common terms reveal what the page is plausibly about.

What You'll Build & Learn

Strip HTML tags, tokenize the visible text into lowercase words, and report the top terms.

- Indexing starts by stripping HTML tags
- Regex extracts lowercase word tokens
- Term counts hint at page topic

🧭 Skill Roadmap

`Skill step:` moves you from recognition toward independent tracing. It builds on Project 12: find the most-linked page. Project 14 changes the surface problem while preserving the underlying method.

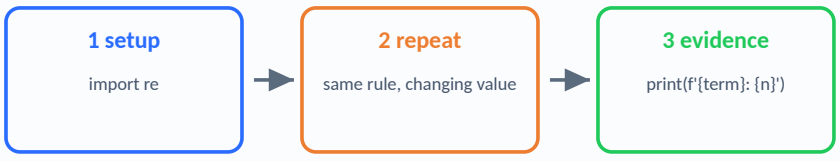
progress check

`Reading target:` tokenize an html body into terms is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- `Writing move:` this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- `Evidence:` `print(f'{term}: {n}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: `setup` creates the facts, `repeat` applies the chapter idea, and `evidence` proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`body = '<h1>Best Running Shoes</h1><p>Running shoes for fast runners. Shoes!</p>'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{term}: {n}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `body = '<h1>Best Running Shoes</h1><p>Running shoes for fast runners. Shoes!</p>'`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Tokenize an HTML Body into Terms: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: re, collections, Counter, body, h1. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: import re | Core: body = '<h1>Best Running Shoes</h1><p>Running shoes for fast runners. S... | Evidence: print(f'{term}: {n}')

🛠️ Reader to Writer Bridge

1. Read: explain tokenize an html body into terms in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 12.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 14 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{term}: {n}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of tokenize an html body into terms and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A regex removes `<...>` tags, then `findall` over lowercased text extracts alphabetic word tokens, which a `Counter` tallies into term frequencies.

```python
import re
from collections import Counter

body = '<h1>Best Running Shoes</h1><p>Running shoes for fast runners. Shoes!
</p>'
text = re.sub(r'<[^>]+>', ' ', body)
tokens = re.findall(r'[a-z]+', text.lower())
counts = Counter(tokens)
for term, n in counts.most_common(3):
    print(f'{term}: {n}')
```

```
output
shoes: 3
running: 2
```

```
best: 1
```

How it works:

1. re.sub replaces every angle-bracket tag with a space, leaving plain text.
2. Lowercasing then re.findall pulls out runs of letters as tokens.
3. A Counter tallies how many times each token appears.
4. most_common(3) prints shoes, running, best with their counts.

Try it:

1. Remove stop words like for and the before counting.
2. Keep digits in tokens for product codes and prices.
3. Weight words inside an h1 tag more heavily.
4. Strip a stop-word list so `for` and `the` do not dominate.

⚠ Common Mistakes

- A greedy `<.*>` would swallow text between two tags; `<[^>]+>` stops at the first `>`.
- Counting before lowercasing makes `Shoes` and `shoes` distinct terms.
- Splitting on whitespace instead of a regex keeps trailing punctuation glued to words.

Skills gained regex tokenizing | HTML stripping | Counter most_common

→ Where You'll Use This: This is the front end of any indexer or on-page SEO tool that needs to know which terms a page actually emphasises.

Mental Model

Tokenizing is shredding a magazine into individual word cards so you can count which topics fill the page.

Project 14 Build an Inverted Index and Query It

The Idea Behind It

An inverted index flips documents inside out, mapping each term to the set of pages containing it, which is what makes search feel instant. This code iterates the document bodies, splits each into words, and adds the page id to that term's set in a defaultdict. A query then becomes a single dictionary lookup returning the matching page ids rather than a scan of every document.

What You'll Build & Learn

Build an inverted index from a few documents and answer a one-word query with a single lookup.

- An inverted index maps terms to pages

- It makes lookups instant
- defaultdict(set) avoids missing-key checks

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 13: tokenize an html body into terms. Project 15 changes the surface problem while preserving the underlying method.

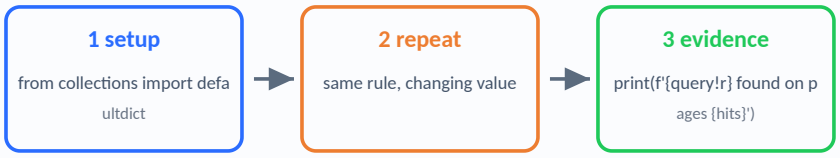
progress check

Reading target: build an inverted index and query it is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{query!r} found on pages {hits}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`for pid, body in docs.items():`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'{query!r} found on pages {hits}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for pid, body in docs.items():`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build an Inverted Index and Query It: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, defaultdict, docs, cheap, red. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: for pid, body in docs.items(): | Evidence: print(f'{query!r} found on pages {hits}')

Reader to Writer Bridge

1. Read: explain build an inverted index and query it in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 13.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 15 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{query!r} found on pages {hits}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build an inverted index and query it and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For each document, every term gains the page id in a `defaultdict(set)`; answering a query is then just reading that term's posting set, sorted for stable output.

```
from collections import defaultdict

docs = {
    1: 'cheap red shoes',
    2: 'red running shoes',
    3: 'blue running socks',
}
index = defaultdict(set)
for pid, body in docs.items():
    for term in body.split():
        index[term].add(pid)
query = 'running'
hits = sorted(index.get(query, set()))
print(f'{query!r} found on pages {hits}')
```

```
output
'running' found on pages [2, 3]
```

How it works:

1. index is a defaultdict of sets, one entry per term.
2. For each document, splitting the body yields its terms.
3. index[term].add(pid) records that this page contains the term.
4. Querying running returns pages 2 and 3 from one lookup.

Try it:

1. Support a two-word AND query by intersecting term sets.
2. Store term positions so you can match exact phrases.
3. Lowercase and de-punctuate tokens before indexing them.
4. Query a term in only one document and confirm a single-page result.

⚠ Common Mistakes

- Using a list instead of a set lets a repeated term add the same page id twice.
- Looking up a missing term with `index[query]` mutates the defaultdict; `.get` avoids that.
- Indexing on raw case so `Shoes` and `shoes` land in different postings.

Skills gained inverted index | defaultdict(set) | query lookup

→ Where You'll Use This: The inverted index is the core data structure of every search engine and site-search feature, turning queries into fast set lookups.

⚡ Mental Model

An inverted index is a book's back-of-book index: instead of reading every page, you jump straight to the pages a word appears on.

Project 15 Rank Pages by Term Frequency

💡 The Idea Behind It

Once you know which pages match a query you must order them, and the crudest ranking simply counts how often the query term appears, rewarding more mentions with a higher position. This code builds a scores dict via str.count for each body, then sorts by negative score with the page id as a tiebreaker so output is stable. The most repetitive page lands on top.

What You'll Build & Learn

Score each page by how many times the query term appears and rank them from most to least.

- Term frequency is a naive ranking
- More mentions means higher rank
- Negative-score sort orders descending

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 14: build an inverted index and query it. Project 16 changes the surface problem while preserving the underlying method.

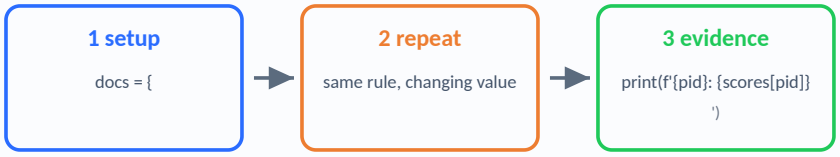
progress check

Reading target: rank pages by term frequency is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `docs = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{pid}: {scores[pid]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`docs = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`'p2': 'running shoes for runners',`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{pid}: {scores[pid]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `'p2': 'running shoes for runners',`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Rank Pages by Term Frequency: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: docs, p1, shoes, running, p2. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: docs = { | Core: 'p2': 'running shoes for runners', | Evidence: print(f'{pid}: {scores[pid]}')

🛠️ Reader to Writer Bridge

1. Read: explain rank pages by term frequency in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 14.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 16 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{pid}: {scores[pid]}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `docs = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of rank pages by term frequency and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Counting the query term per document gives a score, and sorting by negative score with the page id as a tiebreaker yields a deterministic ranked list.

```
docs = {
    'p1': 'shoes shoes running shoes',
    'p2': 'running shoes for runners',
    'p3': 'running running running gear',
}
query = 'running'
scores = {pid: body.split().count(query) for pid, body in docs.items()}
ranked = sorted(scores, key=lambda p: (-scores[p], p))
for pid in ranked:
    print(f'{pid}: {scores[pid]}')
```

```
output
p3: 3
p1: 1
p2: 1
```

How it works:

1. A dict comprehension counts the query word in each document body.
2. sorted ranks pages by negative score, then page id to break ties.
3. p3 with three mentions of running ranks first.
4. p1 and p2 tie at one and order alphabetically.

Try it:

1. Normalise frequency by document length to be fairer.
2. Rank by total occurrences of two query words combined.
3. Print the matching pages with zero hits excluded.
4. Normalize by document length to favour focused pages over long ones.

⚠ Common Mistakes

- Sorting by `scores[p]` ascending puts the worst match first.
- Omitting the page-id tiebreaker makes equal-score ties order unpredictably.

- Using `in` instead of `.count` treats one mention the same as ten.

Skills gained term frequency | sort by key | tie-break ordering

→ Where You'll Use This: Raw term frequency is the baseline ranking that later signals like TF-IDF and BM25 improve upon, and it explains why keyword stuffing once worked.

⚡ Mental Model

Term-frequency ranking is judging relevance by how loudly each page shouts the keyword, with no regard for substance.

Project 16 Classify a User-Agent as Bot or Human

💡 The Idea Behind It

Servers distinguish crawlers from browsers by inspecting the User-Agent header, and spotting marker substrings is the practical test. This code lowercases each agent string and checks whether any of the markers bot, spider, or crawl appears using any() over a tuple. Googlebot and bingbot are flagged as bots while the Firefox and Safari strings pass as human, exactly how a site profiles incoming traffic.

What You'll Build & Learn

Classify each User-Agent string as a bot or a human by scanning for crawler marker substrings.

- User-Agent strings reveal the client
- Marker substrings flag crawlers
- any() tests several markers at once

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 15: rank pages by term frequency. Project 17 changes the surface problem while preserving the underlying method. *progress check*

Reading target: classify a user-agent as bot or human is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

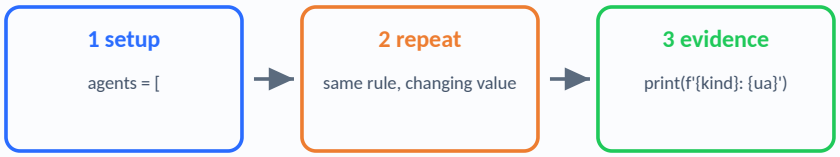
🔎 Logic Lens

- Setup: `agents = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are

discovered.

- Evidence: `print(f'{kind}: {ua}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`agents = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`kind = 'BOT' if any(m in low for m in markers) else 'human'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{kind}: {ua}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `kind = 'BOT' if any(m in low for m in markers) else 'human'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Classify a User-Agent as Bot or Human: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: agents, Mozilla, compatible, Googlebot, Windows. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: agents = [| Core: kind = 'BOT' if any(m in low for m in markers) else 'human' | Evidence: print(f'{kind}: {ua}')

🛠️ Reader to Writer Bridge

1. Read: explain classify a user-agent as bot or human in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 15.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 17 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{kind}: {ua}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `agents = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify a user-agent as bot or human and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Lowercasing the User-Agent and testing for any crawler marker substring (`bot`, `spider`, `crawl`) classifies each visitor with a simple membership check.

```
agents = [
    'Mozilla/5.0 (compatible; Googlebot/2.1)',
    'Mozilla/5.0 (Windows NT 10.0) Gecko Firefox/120',
    'Mozilla/5.0 (compatible; bingbot/2.0)',
    'Mozilla/5.0 (Macintosh) Safari/605',
]
markers = ('bot', 'spider', 'crawl')
for ua in agents:
    low = ua.lower()
    kind = 'BOT' if any(m in low for m in markers) else 'human'
    print(f'{kind}: {ua}')
```

```
output
BOT: Mozilla/5.0 (compatible; Googlebot/2.1)
human: Mozilla/5.0 (Windows NT 10.0) Gecko Firefox/120
BOT: Mozilla/5.0 (compatible; bingbot/2.0)
human: Mozilla/5.0 (Macintosh) Safari/605
```

How it works:

1. markers is a tuple of crawler giveaway substrings.
2. Each agent string is lowercased for case-insensitive matching.
3. any(m in low for m in markers) returns true if any marker appears.
4. Googlebot and bingbot print as BOT, the browser strings as human.

Try it:

1. Identify which specific bot matched, not just BOT.
2. Add fake-bot detection by checking the source IP.
3. Count how many requests came from bots versus humans.
4. Add `'AhrefsBot'` and confirm it is flagged.

⚠ Common Mistakes

- Matching case-sensitively misses `Googlebot` if you only check `googlebot`.
- Checking `== 'bot'` instead of substring membership never matches a real User-Agent.
- A browser string containing 'bot' in a product token would be a false positive to watch for.

Skills gained substring matching | any() predicate | user-agent parsing

→ Where You'll Use This: User-Agent sniffing drives log-file analysis, bot management, and serving crawlers the same content as users to avoid cloaking penalties.

Mental Model

Reading a User-Agent is checking a visitor's badge at the door: the lanyard says whether a robot or a person walked in.

Project 17 Enforce a Per-Minute Crawl Rate Cap

The Idea Behind It

Polite crawlers throttle themselves so they never hammer a server, and you can verify politeness by checking whether any rolling 60-second window breaks a request cap. This code, for each fetch timestamp, counts how many other timestamps fall inside the trailing window, tracking the worst such count. Comparing that peak against the cap of five yields a verdict, and here six fetches in one window flag the crawler as too fast.

What You'll Build & Learn

Scan fetch timestamps for the busiest 60-second window and judge it against a per-minute cap.

- Crawlers throttle to stay polite
- Rolling windows measure request rate
- Peak count tested against the cap

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 16: classify a user-agent as bot or human. Project 18 changes the surface problem while preserving the underlying method.

progress check

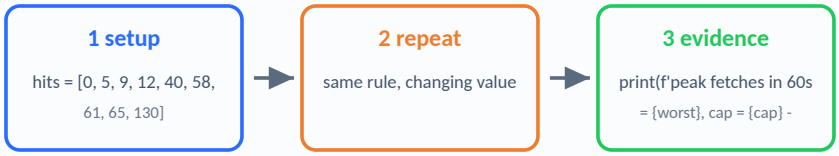
Reading target: enforce a per-minute crawl rate cap is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `hits = [0, 5, 9, 12, 40, 58, 61, 65, 130]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'peak fetches in 60s = {worst}, cap = {cap} -> {verdict}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`hits = [0, 5, 9, 12, 40, 58, 61, 65, 130]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`count = sum(1 for u in hits if t - window < u <= t)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'peak fetches in 60s = {worst}, cap = {cap} -> {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `count = sum(1 for u in hits if t - window < u <= t)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Enforce a Per-Minute Crawl Rate Cap: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: hits, cap, window, worst, enumerate. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: hits = [0, 5, 9, 12, 40, 58, 61, 65, 130] | Core: count = sum(1 for u in hits if t - window < u <= t) | Evidence: print(f'peak fetches in 60s = {worst}, cap = {cap} -> {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain enforce a per-minute crawl rate cap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 16.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 18 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'peak fetches in 60s = {worst}, cap = {cap} -> {verdict}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `hits = [0, 5, 9, 12, 40, 58, 61, 65, 130]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of enforce a per-minute crawl rate cap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For every timestamp, counting how many fetches fall in the trailing 60-second window finds the busiest moment; if that peak exceeds the cap the crawl was impolite.

```
hits = [0, 5, 9, 12, 40, 58, 61, 65, 130]
cap = 5
window = 60
worst = 0
for i, t in enumerate(hits):
```

```
    count = sum(1 for u in hits if t - window < u <= t)
    worst = max(worst, count)
verdict = 'OK' if worst <= cap else 'TOO FAST'
print(f'peak fetches in 60s = {worst}, cap = {cap} -> {verdict}')
```

```
output
peak fetches in 60s = 6, cap = 5 -> TOO FAST
```

How it works:

1. For each timestamp t, count fetches within the trailing 60-second window.
2. The condition t - window < u <= t selects hits ending at t.
3. worst keeps the maximum window count seen across all timestamps.
4. Six exceeds the cap of five, so the verdict prints TOO FAST.

Try it:

1. Change the window to 30 seconds and recheck the peak.
2. Report the timestamp where the busiest window occurs.
3. Slow the schedule until every window obeys the cap.
4. Lower `cap` to 6 and watch the verdict flip to OK.

⚠ Common Mistakes

- Counting fetches per fixed calendar minute misses bursts that straddle a boundary.
- Using `<=` on both window ends double-counts the edge timestamp.
- Comparing the total fetch count to the cap instead of the per-window peak.

Skills gained sliding window | rate limiting | peak tracking

→ Where You'll Use This: Crawl-rate analysis underlies both crawl-delay settings you configure and the server logs you inspect when a bot overloads a site.

Mental Model

It is a bouncer counting how many people entered in the last minute and shouting stop when the door gets too busy.

Project 18 Merge Sitemap and Link Discovery

The Idea Behind It

Crawlers discover URLs two ways, by following links and by reading the sitemap, and the full discovery set is the union of both sources. This code unions a set of link-found URLs with a set of sitemap URLs, then subtracts

links from sitemap to expose URLs the sitemap alone surfaced. Those sitemap-only pages, contact and pricing, would be invisible to a pure link crawl.

What You'll Build & Learn

Combine link-discovered and sitemap URLs, then identify which pages only the sitemap revealed.

- URLs come from links and sitemaps
- Union gives the full discovery set
- Difference finds sitemap-only pages

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 17: enforce a per-minute crawl rate cap. Project 19 changes the surface problem while preserving the underlying method.

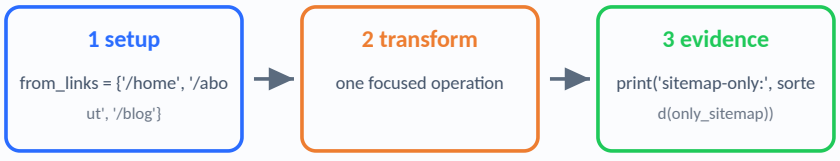
progress check

Reading target: merge sitemap and link discovery is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from_links = {'/home', '/about', '/blog'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is deliberately small: Python stores a value, applies one clear operation, and prints evidence. That tiny chain is the reading pattern larger programs still use.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print('sitemap-only:', sorted(only_sitemap))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from_links = {'/home', '/about', '/blog'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`from_sitemap = {'/home', '/blog', '/pricing', '/contact'}`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print('sitemap-only:', sorted(only_sitemap))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `from_sitemap = {'/home', '/blog', '/pricing', '/contact'}`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Merge Sitemap and Link Discovery: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: from_links, home, about, blog, from_sitemap. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from_links = {'/home', '/about', '/blog'} | Core: from_sitemap = {'/home', '/blog', '/pricing', '/contact'} | Evidence: print('sitemap-only:', sorted(only_sitemap))

Reader to Writer Bridge

1. Read: explain merge sitemap and link discovery in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 17.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 19 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('sitemap-only:', sorted(only_sitemap))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from_links = {'/home', '/about', '/blog'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of merge sitemap and link discovery and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Set union combines both discovery channels, while subtracting the link set from the sitemap set reveals the pages only the sitemap could have surfaced.

```
from_links = {'/home', '/about', '/blog'}
from_sitemap = {'/home', '/blog', '/pricing', '/contact'}
discovered = from_links | from_sitemap
only_sitemap = from_sitemap - from_links
print('total URLs:', len(discovered))
print('sitemap-only:', sorted(only_sitemap))
```

```
output
total URLs: 5
sitemap-only: ['/contact', '/pricing']
```

How it works:

1. from_links and from_sitemap are two sets of discovered paths.
2. The union operator merges them into the complete discovered set.
3. from_sitemap minus from_links isolates sitemap-only URLs.
4. Output reports five total URLs and names contact and pricing as sitemap-only.

Try it:

1. Find URLs in links but absent from the sitemap.
2. Add a third source such as a referral log set.

3. Report the percentage of URLs unique to each source.
4. Add a URL found only by links and confirm it is not sitemap-only.

⚠ Common Mistakes

- Using list concatenation instead of set union leaves duplicate URLs.
- Subtracting in the wrong order (`from_links - from_sitemap`) answers a different question.
- Forgetting to sort the difference makes the printed order nondeterministic.

Skills gained set union | set difference | discovery merging

→ Where You'll Use This: A well-built sitemap surfaces pages that internal links miss, which is exactly how you rescue orphan pages from never being crawled.

Mental Model

Links are word-of-mouth, the sitemap is the official directory: combining both gives you every address in town.

Project 19 Detect a Crawl Trap Cycle

The Idea Behind It

A crawl trap is a loop of links that sends a crawler round forever, and detecting a cycle in the link graph lets you spot the trap before it burns budget. This recursive function carries the path taken so far and returns the looping slice the moment it revisits a node already on that path. Following home through a, b, c uncovers the a, b, c, a cycle.

What You'll Build & Learn

Recursively search the link graph for a cycle and print the looping sequence of pages.

- Crawl traps are link cycles
- Cycles waste crawl budget
- Path tracking detects revisits

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 18: merge sitemap and link discovery. Project 20 changes the surface problem while preserving the underlying method.

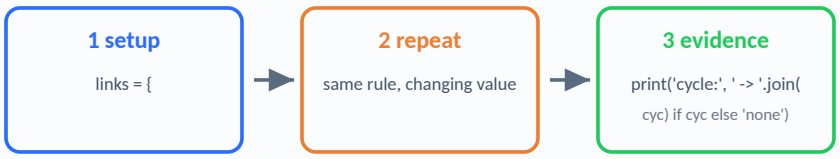
progress check

Reading target: detect a crawl trap cycle is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print('cycle:', ' -> '.join(cyc) if cyc else 'none')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if node in path:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('cycle:', ' -> '.join(cyc) if cyc else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if node in path:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Detect a Crawl Trap Cycle: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: links, home, find_cycle, node, path. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: links = { | Core: if node in path: | Evidence: print('cycle:', ' -> '.join(cyc) if cyc else 'none')

🛠️ Reader to Writer Bridge

1. Read: explain detect a crawl trap cycle in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 18.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 20 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('cycle:', ' -> '.join(cyc) if cyc else 'none')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of detect a crawl trap cycle and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A depth-first walk that carries its current path detects a cycle the moment it revisits a node already on that path, returning the looping segment.

```
links = {
    'home': ['a'],
    'a': ['b'],
    'b': ['c'],
    'c': ['a'],
}

def find_cycle(node, path):
    if node in path:
        return path[path.index(node):] + [node]
    for nxt in links[node]:
        res = find_cycle(nxt, path + [node])
        if res:
            return res
    return None

cyc = find_cycle('home', [])
print('cycle:', ' -> '.join(cyc) if cyc else 'none')
```

```
output
cycle: a -> b -> c -> a
```

How it works:

1. find_cycle takes the current node and the path of ancestors.
2. If the node is already in path, it returns the loop slice from that point.
3. Otherwise it recurses into each child with the node appended to path.
4. Starting at home, it surfaces the cycle a -> b -> c -> a.

Try it:

1. Make the function find all cycles, not the first.
2. Convert it to an iterative search using an explicit stack.
3. Add a branch with no cycle and confirm none is reported.
4. Break the loop by pointing `c` to a dead-end and confirm 'none'.

⚠ Common Mistakes

- Checking a global `visited` set instead of the current path reports a diamond as a false cycle.
- Building `path + [node]` is essential; mutating one shared list corrupts the recursion.
- Deep or wide graphs can hit Python's recursion limit with this naive approach.

Skills gained cycle detection | recursion | path tracking

→ Where You'll Use This: Cycle detection flags infinite loops from calendar widgets, faceted filters, or redirect chains that quietly drain a site's crawl budget.

Mental Model

A crawl trap is a hall of revolving doors: keep pushing forward and you end up exactly where you started, forever.

Project 20 Compute Each Page's Crawl Depth

The Idea Behind It

Crawl depth is the shortest number of clicks from the seed to a page, and BFS computes it naturally because it reaches each page by the fewest hops first. This code seeds a depth dict at zero for home and, during the breadth-first sweep, records each newly discovered child as one deeper than its parent. The result assigns every reachable page a shortest-path distance from the homepage.

What You'll Build & Learn

Compute and print the shortest click-depth from the seed to every reachable page.

- Crawl depth is shortest clicks from seed
- BFS yields shortest-path distances
- Deep pages are crawled less often

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 19: detect a crawl trap cycle. Project 21 changes the surface problem while preserving the underlying method. *progress check*

Reading target: compute each page's crawl depth is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{page}: depth {depth[page]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if nxt not in depth:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{page}: depth {depth[page]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt not in depth:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute Each Page's Crawl Depth: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, depth. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if nxt not in depth: | Evidence: print(f'{page}: depth {depth[page]}')

🛠️ Reader to Writer Bridge

1. Read: explain compute each page's crawl depth in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 19.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 21 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{page}: depth {depth[page]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of compute each page's crawl depth and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

BFS naturally discovers each node by its shortest path, so recording `depth[neighbour] = depth[current] + 1` on first sight gives every page its true crawl depth.

```
from collections import deque

links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c', 'd'],
    'c': ['e'],
    'd': [], 'e': [],
}
depth = {'home': 0}
q = deque(['home'])
while q:
    page = q.popleft()
    for nxt in links[page]:
        if nxt not in depth:
            depth[nxt] = depth[page] + 1
            q.append(nxt)
for page in sorted(depth):
    print(f'{page}: depth {depth[page]}')
```

```
output
a: depth 1
b: depth 1
c: depth 2
d: depth 2
e: depth 3
home: depth 0
```

How it works:

1. depth starts with home at zero and the queue holds home.
2. Each child not yet in depth gets its parent's depth plus one.
3. Because BFS reaches pages by fewest hops, the first depth recorded is shortest.
4. Sorted output shows home at zero rising to e at depth three.

Try it:

1. Flag every page deeper than two clicks.
2. Report the average depth across all pages.
3. Add a shortcut link and watch a page's depth drop.

4. Add a shortcut link from `home` to `e` and watch its depth drop.

⚠ Common Mistakes

- Using DFS instead of BFS records a path length, not the shortest one.
- Overwriting a depth when a page is reached again by a longer route inflates it.
- Seeding the queue without putting the seed in `depth` at depth 0 desynchronises the two.

Skills gained BFS shortest path | depth labelling | graph distance

→ Where You'll Use This: Click depth is a practical SEO health metric: keeping important pages within three clicks of the homepage helps them get crawled and indexed.

Mental Model

Crawl depth is how many doors you open from the lobby to reach a room: the deeper it is, the less often anyone visits.

Project 21 Apply Robots Allow and Disallow Rules

The Idea Behind It

A robots file tells crawlers which paths they may fetch, and when rules overlap the longest-matching prefix wins, letting a specific Disallow carve a hole out of a broad Allow. This code scans every rule, keeping the one whose prefix both matches the path and is at least as long as the current best. The final rule's action decides whether each URL is crawlable.

What You'll Build & Learn

Apply robots Allow and Disallow rules by longest-prefix match to decide if each URL is crawlable.

- Robots rules gate crawlable paths
- The longest matching prefix wins
- Specific Disallow beats broad Allow

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 20: compute each page's crawl depth. Project 22 changes the surface problem while preserving the underlying method.

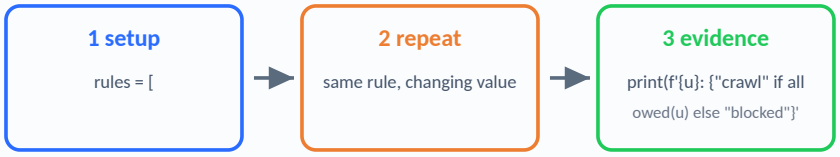
progress check

Reading target: apply robots allow and disallow rules is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `rules = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{u}: {"crawl" if allowed(u) else "blocked"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`rules = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if path.startswith(prefix) and len(prefix) >= len(best[1]):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{u}: {"crawl" if allowed(u) else "blocked"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if path.startswith(prefix) and len(prefix) >= len(best[1]):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Apply Robots Allow and Disallow Rules: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: rules, Disallow, admin, cart, Allow. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: rules = [| Core: if path.startswith(prefix) and len(prefix) >= len(best[1]): | Evidence: print(f'{u}: {"crawl" if allowed(u) else "blocked"}')

🛠️ Reader to Writer Bridge

1. Read: explain apply robots allow and disallow rules in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 20.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 22 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{u}: {"crawl" if allowed(u) else "blocked"}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `rules = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of apply robots allow and disallow rules and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

For each URL the rule with the longest matching prefix decides access, mirroring the real longest-match precedence of robots directives.

```
rules = [
    ('Disallow', '/admin'),
    ('Disallow', '/cart'),
    ('Allow', '/'),
]
urls = ['/', '/blog', '/admin/login', '/cart', '/products']

def allowed(path):
    best = ('Allow', '')
    for action, prefix in rules:
        if path.startswith(prefix) and len(prefix) >= len(best[1]):
            best = (action, prefix)
    return best[0] == 'Allow'

for u in urls:
    print(f'{u}: {"crawl" if allowed(u) else "blocked"}')
```

```
output
/: crawl
/blog: crawl
/admin/login: blocked
/cart: blocked
/products: crawl
```

How it works:

1. allowed starts with a default Allow on an empty prefix.
2. It scans rules, keeping any whose prefix matches and is at least as long.
3. The longest-matching rule's action becomes the verdict for the path.
4. /admin/login and /cart hit Disallow rules and print as blocked.

Try it:

1. Add wildcard pattern support like /private/*.
2. Handle a path matched by no rule at all.
3. Report which rule won for each tested URL.

4. Add `('Allow', '/cart/gift')` and confirm that sub-path becomes crawlable.

⚠ Common Mistakes

- First-match-wins instead of longest-match gives wrong verdicts when rules overlap.
- Comparing prefix length without `startswith` matches unrelated paths.
- Treating an empty default as Disallow blocks everything not explicitly allowed.

Skills gained longest-prefix match | robots rules | access control

→ Where You'll Use This: This is the exact precedence logic behind robots.txt parsing, which determines whether Googlebot is even permitted to request a given URL.

Mental Model

Robots rules are zoning laws: a broad allow covers the whole city, but a precise no-entry sign overrides it on one street.

Project 22 Fold Duplicates with Canonical Tags

The Idea Behind It

Tracking parameters and trailing slashes spawn many URLs for one page, and a canonical tag tells the engine the single address to index while folding the rest away. This code reads each page's declared canonical, builds a set of those canonicals to find the truly unique pages, and subtracts that from the total to count how many duplicates were absorbed. Five URLs collapse to three indexable pages.

What You'll Build & Learn

Use declared canonical tags to find the unique indexable pages and count the duplicates folded away.

- Many URLs can map to one page
- Canonical tags pick the index URL
- Set of canonicals counts unique pages

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 21: apply robots allow and disallow rules. Project 23 changes the surface problem while preserving the underlying method.

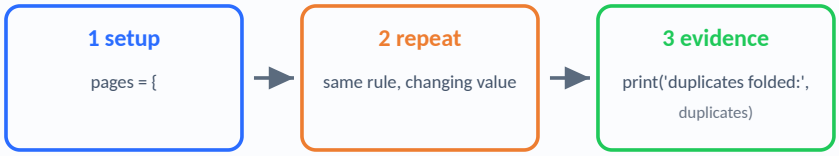
progress check

Reading target: fold duplicates with canonical tags is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print('duplicates folded:', duplicates)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`indexable = sorted({p['canonical'] for p in pages.values()})`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('duplicates folded:', duplicates)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `indexable = sorted({p['canonical'] for p in pages.values()})`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Fold Duplicates with Canonical Tags: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: pages, home, canonical, ref, ad. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: indexable = sorted({p['canonical'] for p in pages.values()}) | Evidence: print('duplicates folded:', duplicates)

🛠️ Reader to Writer Bridge

1. Read: explain fold duplicates with canonical tags in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 21.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 23 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('duplicates folded:', duplicates)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of fold duplicates with canonical tags and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Collecting the distinct canonical values gives the set of pages actually indexed; the gap between total URLs and canonicals is the count of duplicates folded.

```
pages = {
    '/home': {'canonical': '/home'},
    '/home?ref=ad': {'canonical': '/home'},
    '/blog': {'canonical': '/blog'},
    '/blog/': {'canonical': '/blog'},
    '/about': {'canonical': '/about'},
}
indexable = sorted({p['canonical'] for p in pages.values()})
duplicates = len(pages) - len(indexable)
print('unique pages indexed:', indexable)
print('duplicates folded:', duplicates)
```

```
output
unique pages indexed: ['/about', '/blog', '/home']
duplicates folded: 2
```

How it works:

1. Each page in the dict declares a canonical URL.
2. A set comprehension collects the distinct canonical values.
3. len(pages) minus that set's length counts folded duplicates.
4. Three unique pages are indexed and two duplicates are folded.

Try it:

1. Group every source URL under its canonical target.
2. Flag a page whose canonical points elsewhere unexpectedly.
3. Detect a canonical that points to a noindexed page.
4. Point `/blog/`'s canonical at `/home` and watch `/blog` vanish.

⚠ Common Mistakes

- Counting raw URLs as indexed pages over-reports the index size.
- Forgetting to dedupe canonicals double-counts a page reachable by two URLs.
- A page pointing its canonical at a different page would silently disappear from the index.

Skills gained canonicalization | set comprehension | duplicate folding

→ Where You'll Use This: Canonical handling decides which of many parameterised or paginated URLs Google indexes, a constant concern on e-commerce and faceted sites.

Mental Model

Canonical tags are mail forwarding: many addresses can reach one person, but the engine only ever delivers to the home address.

Project 23 Route Pages by HTTP Status Code

The Idea Behind It

A crawler's next move hinges on the HTTP status a URL returns: 200 means index, a 3xx means follow the redirect, and 4xx or 5xx means drop. This code iterates page-and-code pairs and uses an if/elif chain to assign each a verdict, with the 300-to-399 range explicitly tested for redirects. The status code, not the content, drives the entire indexing decision here.

What You'll Build & Learn

Route each page to index, follow-redirect, or drop based purely on its HTTP status code.

- Status code drives indexing decisions
- 200 indexes, 3xx redirects, 4xx and 5xx drop
- A range test classifies redirects

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 22: fold duplicates with canonical tags. Project 24 changes the surface problem while preserving the underlying method.

progress check

Reading target: route pages by http status code is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print(f'{code} {path}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if code == 200:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{code} {path}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if code == 200:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Route Pages by HTTP Status Code: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: pages, home, old, missing, blog. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = [| Core: if code == 200: | Evidence: print(f'{code} {path}: {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain route pages by http status code in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 22.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 24 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code} {path}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of route pages by http status code and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Bucketing each status into 200, the 300 range, or everything else maps the numeric code to the crawler's action, exactly as a real fetch pipeline branches.

```
pages = [
    ('/home', 200),
    ('/old', 301),
    ('/missing', 404),
    ('/blog', 200),
    ('/down', 503),
]
for path, code in pages:
    if code == 200:
        verdict = 'index'
    elif 300 <= code < 400:
        verdict = 'follow redirect'
    else:
        verdict = 'drop'
    print(f'{code} {path}: {verdict}')
```

```
output
200 /home: index
301 /old: follow redirect
404 /missing: drop
200 /blog: index
503 /down: drop
```

How it works:

1. The loop unpacks each path and its HTTP status code.
2. A code of 200 sets the verdict to index.
3. A code in the 300-399 range sets follow redirect.
4. Everything else, including 404 and 503, sets drop.

Try it:

1. Split 4xx from 5xx with distinct verdicts.
2. Treat 503 as retry-later instead of drop.
3. Count how many pages fall into each bucket.
4. Add a `410` (gone) and decide whether to drop it permanently.

⚠ Common Mistakes

- Testing `code == 300` instead of the `300 <= code < 400` range misses 301 and 302.
- Treating a 5xx server error like a 4xx not-found ignores that it may recover later.
- Indexing a 204 (no content) as if it were a 200 with a body.

Skills gained status routing | if/elif chain | range testing

→ Where You'll Use This: Status-code triage is the heart of crawl-error reporting: it explains why pages get indexed, redirected, or dropped from search results.

Mental Model

The status code is a traffic light at each URL: green go, amber detour, red stop and turn back.

Project 24 Honour the Noindex Directive

The Idea Behind It

A page can be crawled yet explicitly asked not to be indexed through a noindex meta directive, so search-result and print pages typically carry it to stay out of the index. This code reads each page's noindex flag and partitions the URLs with two list comprehensions: one keeps pages where the flag is false, the other collects the blocked ones. Crawlability and indexability are clearly separated.

What You'll Build & Learn

Split pages into indexable and noindexed sets based on each page's noindex meta directive.

- Noindex blocks indexing, not crawling
- Crawlable is not the same as indexable
- List comprehensions partition the pages

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 23: route pages by http status code. Project 25 changes the surface problem while preserving the underlying method.

progress check

Reading target: honour the noindex directive is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print('noindex:', sorted(blocked))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`indexable = [p for p, m in pages.items() if not m['noindex']]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('noindex:', sorted(blocked))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `indexable = [p for p, m in pages.items() if not m['noindex']]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Honour the Noindex Directive: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: pages, home, noindex, search, results. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: indexable = [p for p, m in pages.items() if not m['noindex']] | Evidence: print('noindex:', sorted(blocked))

🛠️ Reader to Writer Bridge

1. Read: explain honour the noindex directive in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 23.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 25 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('noindex:', sorted(blocked))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of honour the noindex directive and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Partitioning pages on their `noindex` flag separates the indexable set from the deliberately excluded one, with sorting keeping both lists deterministic.

```
pages = {
    '/home': {'noindex': False},
    '/search-results': {'noindex': True},
    '/blog': {'noindex': False},
    '/print-view': {'noindex': True},
}
indexable = [p for p, m in pages.items() if not m['noindex']]
blocked = [p for p, m in pages.items() if m['noindex']]
print('indexable:', sorted(indexable))
print('noindex:', sorted(blocked))
```

```
output
indexable: ['/blog', '/home']
noindex: ['/print-view', '/search-results']
```

How it works:

1. Each page maps to a metadata dict carrying a noindex boolean.
2. One comprehension keeps pages where noindex is false.
3. Another collects the pages where noindex is true.
4. Sorted output lists blog and home as indexable, the rest as noindex.

Try it:

1. Add a nofollow flag and handle it separately.
2. Warn when a noindexed page still receives internal links.
3. Count what share of pages carry the noindex directive.
4. Flip `/blog` to noindex and watch it move lists.

⚠ Common Mistakes

- Confusing noindex (keep crawling, do not index) with a robots Disallow (do not crawl).
- Treating a missing `noindex` key as truthy and dropping good pages.
- Printing unsorted sets so the two lists vary run to run.

Skills gained list comprehension directive handling set partitioning

→ Where You'll Use This: Honouring noindex keeps thin or duplicate pages like search results and filters out of the index without blocking crawlers from reaching them.

Mental Model

Noindex is a guest who attends the party but asks not to appear in any photos: present, yet absent from the album.

Project 25 Follow a Redirect Chain to the Target

The Idea Behind It

URLs often redirect through several hops before landing on the real page, so a crawler walks the chain to find the final destination and counts the steps. This code follows a redirect dict from a start URL, appending each new target to a chain list and incrementing a hop counter, with a guard of ten to prevent runaway loops. The chain resolves cleanly to /final after three hops.

What You'll Build & Learn

Follow a redirect chain from a start URL to its final target and count the number of hops.

- Redirects can chain several hops
- Crawlers follow to the final URL
- A hop cap guards against loops

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 24: honour the noindex directive. Project 26 changes the surface problem while preserving the underlying method. *progress check*

Reading target: follow a redirect chain to the target is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `redirects = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

- Evidence: `print('hops:', hops)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`redirects = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`while cur in redirects and hops < 10:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('hops:', hops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `while cur in redirects and hops < 10:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Follow a Redirect Chain to the Target: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: redirects, start, chain, cur, hops. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: redirects = { | Core: while cur in redirects and hops < 10: | Evidence: print('hops:', hops)

🛠️ Reader to Writer Bridge

1. Read: explain follow a redirect chain to the target in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 24.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 26 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('hops:', hops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `redirects = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of follow a redirect chain to the target and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Walking the redirect map until a URL has no further target, with a hop guard, resolves the chain and counts how many redirects were traversed.

```
redirects = {
    '/a': '/b',
    '/b': '/c',
    '/c': '/final',
}
start = '/a'
chain, cur, hops = [start], start, 0
while cur in redirects and hops < 10:
    cur = redirects[cur]
    chain.append(cur)
    hops += 1
print(' -> '.join(chain))
print('hops:', hops)
```

```
output
/a -> /b -> /c -> /final
hops: 3
```

How it works:

1. chain begins with the start URL and hops starts at zero.
2. While the current URL has a redirect and hops is under ten, follow it.
3. Each new target is appended to chain and hops increments.
4. The loop ends at /final after three hops, both printed.

Try it:

1. Flag chains longer than two hops as inefficient.
2. Print the destination only, hiding the intermediate URLs.
3. Distinguish 301 permanent from 302 temporary redirects.
4. Add another hop `/final -> /home` and watch the chain lengthen.

⚠ Common Mistakes

- No hop limit means a looping chain spins forever.
- Recording the chain before appending the next hop drops the final URL.
- Counting visited URLs instead of redirects overstates the hop count by one.

Skills gained chain following | loop guard | hop counting

→ Where You'll Use This: Tracing redirect chains diagnoses crawl waste and lost link equity, since long chains slow crawlers and dilute the signal that reaches the target.

Mental Model

Following redirects is forwarded mail that gets forwarded again: you keep chasing the new address until a letter finally arrives.

Project 26 Detect a Redirect Loop

The Idea Behind It

If redirects form a cycle a crawler would loop forever fetching the same URLs, so tracking visited addresses lets you stop and flag the loop. This code follows the redirect dict but adds each URL to a seen set first; the moment it revisits a URL already in seen it sets a loop flag and breaks. The /x, /y, /z cycle is caught after three hops rather than spinning endlessly.

What You'll Build & Learn

Detect a redirect cycle by tracking visited URLs, stopping and flagging the loop instead of looping forever.

- Redirect cycles trap crawlers
- A seen set detects revisits
- break stops the infinite loop

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 25: follow a redirect chain to the target. Project 27 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a redirect loop is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

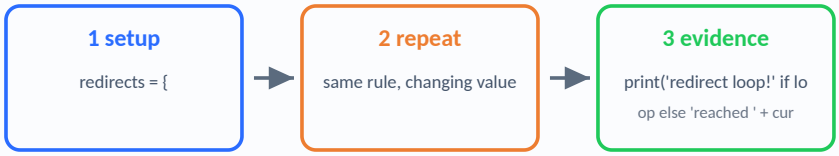
- Setup: `redirects = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print('redirect loop!' if loop else 'reached ' + cur, 'after', hops, 'hops')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`redirects = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if cur in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('redirect loop!' if loop else 'reached ' + cur, 'after', hops, 'hops')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if cur in seen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Redirect Loop: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: redirects, start, cur, seen, hops. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: redirects = { | Core: if cur in seen: | Evidence: print('redirect loop!' if loop else 'reached ' + cur, 'after', hops, 'h...

🛠️ Reader to Writer Bridge

1. Read: explain detect a redirect loop in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 25.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 27 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('redirect loop!' if loop else 'reached ' + cur, 'after', hops, 'hops')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `redirects = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a redirect loop and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Adding each URL to a `seen` set before stepping to its target means re-encountering a seen URL proves a loop, which the code reports rather than looping.

```
redirects = {
    '/x': '/y',
    '/y': '/z',
    '/z': '/x',
}
```

```
start, cur = '/x', '/x'
seen, hops = set(), 0
loop = False
while cur in redirects:
    if cur in seen:
        loop = True
        break
    seen.add(cur)
    cur = redirects[cur]
    hops += 1
print('redirect loop!' if loop else 'reached ' + cur, 'after', hops, 'hops')
```

```
output
redirect loop! after 3 hops
```

How it works:

1. start and cur begin at /x with an empty seen set.
2. Before following, the loop checks if cur is already in seen.
3. A revisit sets loop True and breaks out immediately.
4. Otherwise cur is recorded, the redirect is followed, and hops increments.

Try it:

1. Print the exact URL where the loop closes.
2. Combine the cap guard with the seen-set check.
3. Add a chain that resolves cleanly and confirm no loop.
4. Break the loop by pointing `/z` to `/done` and confirm a clean finish.

⚠ Common Mistakes

- Checking `seen` after advancing misses the URL that closes the loop.
- Using the hop counter alone as a guard hides the fact that it is a loop, not just a long chain.
- Resetting `seen` inside the loop defeats the detection entirely.

Skills gained cycle detection | visited set | loop guard

→ Where You'll Use This: Redirect-loop detection catches the misconfigurations that return ERR_TOO_MANY_REDIRECTS to users and silently waste a crawler's budget.

Mental Model

A redirect loop is a roundabout with no exit: tracking where you have been is the only way to notice you keep passing the same sign.

Project 27 Rank Pages with a PageRank Iteration

The Idea Behind It

PageRank treats each link as a vote and spreads importance around the graph until the scores settle, and a few damped iterations approximate that steady state. This code seeds every page with equal rank, then repeatedly distributes each page's rank across its outbound links, scaled by a 0.85 damping factor with a 0.15 teleport baseline. After twenty rounds the ranks converge, with the well-linked pages scoring highest.

What You'll Build & Learn

Approximate PageRank by iterating link-vote distribution with damping until scores stabilise.

- Links are votes that spread importance
- Damping models random teleportation
- Iteration converges to steady ranks

Skill Roadmap

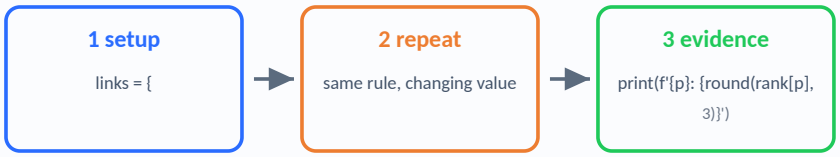
Skill step: pushes the same idea into a less automatic situation. It builds on Project 26: detect a redirect loop. Project 28 changes the surface problem while preserving the underlying method. *progress check*

Reading target: rank pages with a pagerank iteration is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{p}: {round(rank[p], 3)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`rank = {p: 1.0 / N for p in links}`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{p}: {round(rank[p], 3)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rank = {p: 1.0 / N for p in links}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Rank Pages with a PageRank Iteration: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: links, home, len, rank, range. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: links = { | Core: rank = {p: 1.0 / N for p in links} | Evidence: print(f'{p}: {round(rank[p], 3)}')

🛠️ Reader to Writer Bridge

1. Read: explain rank pages with a pagerank iteration in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 26.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 28 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{p}: {round(rank[p], 3)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of rank pages with a pagerank iteration and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each round redistributes 85% of every page's rank evenly across its out-links plus a 15% uniform teleport; repeating converges toward the stationary PageRank values.

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c'],
    'c': ['home'],
}
N = len(links)
```

```
rank = {p: 1.0 / N for p in links}
for _ in range(20):
    nxt = {p: 0.15 / N for p in links}
    for p, outs in links.items():
        share = rank[p] / len(outs)
        for t in outs:
            nxt[t] += 0.85 * share
    rank = nxt
for p in sorted(rank):
    print(f'{p}: {round(rank[p], 3)}')
```

```
output
a: 0.172
b: 0.172
c: 0.33
home: 0.326
```

How it works:

1. Every page starts with rank 1/N, an equal share.
2. Each iteration seeds nxt with the 0.15/N teleport baseline.
3. A page's rank, split across its outbound links, is added scaled by 0.85.
4. After twenty rounds, c and home score highest as central pages.

Try it:

1. Print the rank after each iteration to watch it converge.
2. Handle a dangling page with no outbound links.
3. Change the damping factor and observe the effect.
4. Run only 2 iterations and watch the ranks before they settle.

⚠ Common Mistakes

- Updating `rank` in place mid-round uses half-new values and breaks convergence.
- A page with zero out-links (a dangling node) would leak rank and needs special handling.
- Printing raw floats without `round` gives noisy, nondeterministic-looking output.

Skills gained PageRank iteration | damping factor | rank distribution

→ Where You'll Use This: PageRank's link-vote model still underlies how authority flows through internal links and backlinks, shaping which pages an engine trusts most.

Mental Model

PageRank is reputation by endorsement: you are trusted in proportion to how much trusted people vouch for you, settling over many rounds.

Project 28 Score Pages by Query-Term Coverage

The Idea Behind It

A multi-word query rewards pages that contain more of its words, so coverage, the fraction of query terms present, is a sharper relevance signal than raw frequency. This code splits the query into a term set, then for each document builds a set of its words and intersects the two, scoring by how many query terms overlap. Sorting by coverage ranks the pages that match more of the query higher.

What You'll Build & Learn

Score each page by how many distinct query terms it contains and rank by that coverage.

- Coverage counts query terms present
- Set intersection measures overlap
- More terms matched ranks higher

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 27: rank pages with a pagerank iteration. Project 29 changes the surface problem while preserving the underlying method.

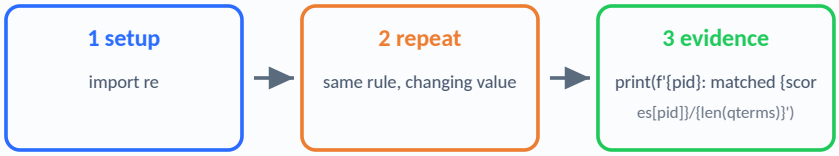
progress check

Reading target: score pages by query-term coverage is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as `data -> transformation -> evidence`, not as isolated library calls.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{pid}: matched {scores[pid]}/{len(qterms)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`for pid, body in docs.items():`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. slow line
3. Check the evidence	`print(f'{pid}: matched {scores[pid]}/{len(qterms)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for pid, body in docs.items():`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score Pages by Query-Term Coverage: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: re, docs, p1, red, running. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: for pid, body in docs.items(): | Evidence: print(f'{pid}: matched {scores[pid]}/{len(qterms)}')

🛠️ Reader to Writer Bridge

1. Read: explain score pages by query-term coverage in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 27.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 29 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{pid}: matched {scores[pid]}/{len(qterms)}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score pages by query-term coverage and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Intersecting each document's term set with the query's term set counts how many query words a page covers; ranking by that coverage orders the most-on-topic pages first.

```
import re

docs = {
    'p1': 'red running shoes',
```

```
    'p2': 'red shoes sale',
    'p3': 'blue running shoes',
}
query = 'running shoes'
qterms = set(query.split())
scores = {}
for pid, body in docs.items():
    terms = set(re.findall(r'[a-z]+', body))
    scores[pid] = len(qterms & terms)
ranked = sorted(scores, key=lambda p: (-scores[p], p))
for pid in ranked:
    print(f'{pid}: matched {scores[pid]}/{len(qterms)}')
```

```
output
p1: matched 2/2
p3: matched 2/2
p2: matched 1/2
```

How it works:

1. The query splits into qterms, a set of its words.
2. Each document body becomes a set via re.findall of letter runs.
3. Intersecting qterms with the document's words and taking len gives the score.
4. p1 and p3 match both terms and outrank p2's single match.

Try it:

1. Break coverage ties using term frequency.
2. Report each page's matched terms, not just the count.
3. Lowercase and de-duplicate the query terms first.
4. Add a page containing both query words once each and confirm full coverage.

⚠ Common Mistakes

- Using term frequency instead of set intersection rewards repetition over coverage.
- Forgetting to lowercase or regex-tokenize lets punctuation hide a match.
- Omitting the page-id tiebreaker makes equal-coverage ties nondeterministic.

Skills gained set intersection | coverage scoring | query matching

→ Where You'll Use This: Term-coverage scoring reflects why pages addressing more of a query's intent rank above pages that merely repeat one keyword many times.

Mental Model

Coverage is grading an answer by how many of the question's points it addresses, not by how often it repeats one of them.

Project 29 Weight a Term with TF-IDF

The Idea Behind It

Counting a term is not enough because a word in every page tells you nothing, so TF-IDF multiplies term frequency by inverse document frequency to reward rare, distinctive words. This code first builds a document-frequency table, then computes the idf for apple as the log of N over its df, and finally scores each page by tf times idf. Pages where apple is both frequent and distinctive score highest.

What You'll Build & Learn

Weight a term per page with TF-IDF and rank the pages by the resulting score.

- Common words carry little signal
- IDF rewards rare distinctive terms
- TF-IDF is frequency times inverse document frequency

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 28: score pages by query-term coverage. Project 30 changes the surface problem while preserving the underlying method.

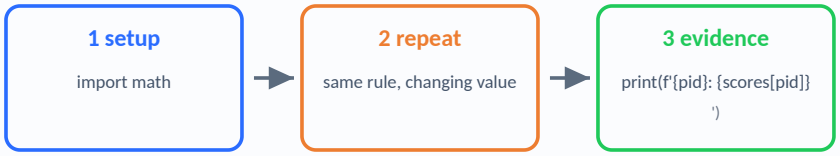
progress check

Reading target: weight a term with tf-idf is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import math` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print(f'{pid}: {scores[pid]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import math`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`for body in docs.values():`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{pid}: {scores[pid]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for body in docs.values():`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Weight a Term with TF-IDF: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: math, docs, p1, apple, banana. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import math | Core: for body in docs.values(): | Evidence: print(f'{pid}: {scores[pid]}')

🛠️ Reader to Writer Bridge

1. Read: explain weight a term with tf-idf in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 28.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 30 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{pid}: {scores[pid]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import math` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of weight a term with tf-idf and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

IDF is `log(N / document_frequency)`, so a term in fewer documents weighs more; multiplying by per-document term frequency ranks pages by distinctive relevance.

```
import math

docs = {
    'p1': 'apple apple banana',
    'p2': 'banana cherry cherry cherry',
    'p3': 'apple cherry',
}
N = len(docs)
```

```
df = {}
for body in docs.values():
    for term in set(body.split()):
        df[term] = df.get(term, 0) + 1
term = 'apple'
idf = math.log(N / df[term])
scores = {}
for pid, body in docs.items():
    tf = body.split().count(term)
    scores[pid] = round(tf * idf, 3)
for pid in sorted(scores, key=lambda p: (-scores[p], p)):
    print(f'{pid}: {scores[pid]}')
```

```
output
p1: 0.811
p3: 0.405
p2: 0.0
```

How it works:

1. df counts how many documents contain each unique term.
2. idf for apple is the log of N divided by its document frequency.
3. Each page's tf is its count of apple, multiplied by idf for the score.
4. p1 scores highest, p3 lower, and p2 zero since it lacks apple.

Try it:

1. Compute TF-IDF for every term, not just apple.
2. Add smoothing so a term in all docs is not exactly zero.
3. Normalise term frequency by document length first.
4. Score `"banana"` (in two docs) and compare its weight to `"apple"`.

⚠ Common Mistakes

- Computing document frequency from raw tokens (not a per-doc set) overcounts repeats.
- A term present in every document gives IDF 0, zeroing its score, which surprises beginners.
- Using `log10` versus natural `log` silently changes every score.

Skills gained TF-IDF weighting | inverse document frequency | log scaling

→ Where You'll Use This: TF-IDF is the classic relevance weighting behind on-page optimisation and content-gap analysis, favouring distinctive terms over ubiquitous filler.

Mental Model

TF-IDF is valuing a fingerprint: a feature shared by everyone identifies no one, but a rare distinctive trait pins down the page.

Project 30 Run the Full Crawl Pipeline Report

The Idea Behind It

This capstone ties the chapter together by running a miniature crawl audit in one pass: from a seed it crawls the link graph with BFS, records every page's depth, and reports coverage, deepest page, and orphans. The depth dict doubles as both the visited set and the distance map, while the full page inventory minus the crawled set exposes the orphan ghost. It is the skeleton of a real audit.

What You'll Build & Learn

Run a combined crawl pipeline that reports coverage, maximum depth, and orphan pages in a single pass.

- A crawl audit fuses several signals
- Depth dict doubles as the visited set
- Coverage, depth, and orphans in one pass

Skill Roadmap

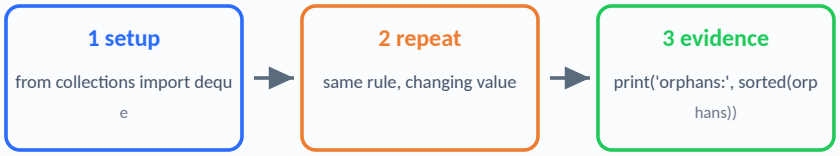
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 29: weight a term with tf-idf. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run the full crawl pipeline report is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import deque` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
- Evidence: `print('orphans:', sorted(orphans))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import deque`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
2. Apply the rule	`if nxt not in depth:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('orphans:', sorted(orphans))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: How Search Engines Work becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt not in depth:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run the Full Crawl Pipeline Report: connect the surface task to the chapter skill, crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.
Names	Watch these names first: collections, deque, links, home, blog. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import deque | Core: if nxt not in depth: | Evidence: print('orphans:', sorted(orphans))

🛠️ Reader to Writer Bridge

1. Read: explain run the full crawl pipeline report in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 29.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('orphans:', sorted(orphans))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import deque` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run the full crawl pipeline report and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

One BFS produces the depth map, and set arithmetic over all known pages yields coverage and orphans, summarising crawl health from a single traversal.

```
from collections import deque

links = {
    'home': ['blog', 'shop'],
    'blog': ['post1', 'post2'],
    'shop': ['item1'],
    'post1': [], 'post2': [], 'item1': [],
```

```
    'ghost': [],
}
seed = 'home'
depth, q = {seed: 0}, deque([seed])
while q:
    p = q.popleft()
    for nxt in links[p]:
        if nxt not in depth:
            depth[nxt] = depth[p] + 1
            q.append(nxt)
crawled = set(depth)
orphans = set(links) - crawled
print('seed:', seed)
print('crawled:', len(crawled), 'of', len(links))
print('max depth:', max(depth.values()))
print('orphans:', sorted(orphans))
```

```
output
seed: home
crawled: 6 of 7
max depth: 2
orphans: ['ghost']
```

How it works:

1. A BFS from the seed fills depth with each page's distance from home.
2. crawled is the set of keys in depth, the pages actually reached.
3. set(links) minus crawled exposes the orphan ghost.
4. The report prints coverage 6 of 7, max depth 2, and the orphan list.

Try it:

1. Add average depth to the report alongside the maximum.
2. Include each crawled page's depth in the output.
3. Feed orphans back in via a sitemap and re-audit.
4. Link `ghost` from `shop` and watch coverage hit 100%.

⚠ Common Mistakes

- Computing orphans before the BFS finishes reports pages still in the queue as unreachable.
- Calling `max` on an empty depth map (no pages) would raise; the seed guarantees one entry here.
- Counting `len(depth)` as total pages instead of crawled pages hides the orphans.

Skills gained crawl audit | BFS pipeline | orphan detection

→ **Where You'll Use This:** This is the blueprint of a production site-crawl tool: one traversal yields the coverage, depth, and orphan metrics that drive technical-SEO reports.

Mental Model

It is a one-walk building inspection: you tour from the entrance, note how deep each room sits, and list any room no hallway reaches.

Chapter Progress Checkpoint

What this chapter added: after 30 mini-projects, How Search Engines Work should feel like a reusable Python move: crawl, index and rank — what a bot sees, crawl budget, and how urls are discovered.

chapter checkpoint

1 Read
Open any project from Projects 1-30 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 2 (robots.txt and Crawl Directives) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

8 questions · 4 categories · graded

2 Recall | 3 Meaning | 2 Structure | 1 Transfer

1.1.Q1 Recall MEDIUM — Turn back to Project 22 (Fold Duplicates with Canonical Tags). Without running it, what exactly does it print?

1.1.Q2 Meaning MEDIUM — Turn back to Project 1 (Model a Tiny Web as a Link Graph). Without running it, what exactly does it print?

1.1.Q3 Structure MEDIUM — Turn back to Project 18 (Merge Sitemap and Link Discovery). Without running it, what exactly does it print?

1.1.Q4 Transfer MEDIUM — Turn back to Project 24 (Honour the Noindex Directive). Without running it, what exactly does it print?

1.1.Q5 Recall EASY A search engine performs three phases in order. What are they?

- A Rank, crawl, index
- B Crawl, index, rank
- C Index, crawl, rank
- D Crawl, rank, index

1.1.Q6 Meaning MEDIUM A page exists and has great content but no other page links to it and it is not in any sitemap. What is most likely true?

- A It will rank #1
- B It may never be crawled or indexed (orphan page)
- C It is automatically indexed
- D It is a duplicate

1.1.Q7 Structure EASY Why does wasting crawl budget on duplicate URLs hurt SEO?

- A Duplicates are illegal
- B Fetches spent re-crawling duplicates are not spent on unique, important pages, delaying their indexing
- C It uses more disk on your server
- D It changes your domain authority

1.1.Q8 Meaning EASY In an inverted index, what does the entry for the term "audit" store?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 1 – How Search Engines Work

1.1.Q1. `unique pages indexed: ['/about', '/blog', '/home']` then `duplicates folded: 2`

1.1.Q2. `pages=5 links=6`

1.1.Q3. `total URLs: 5` then `sitemap-only: ['/contact', '/pricing']`

1.1.Q4. `indexable: ['/blog', '/home']` then `noindex: ['/print-view', '/search-results']`

1.1.Q5. **B** crawl (discover), index (store/understand), then rank (order for a query).

1.1.Q6. **B** with no inbound edge and no sitemap entry, the crawler has no way to discover it.

1.1.Q7. **B** the budget is finite, so junk fetches steal from valuable ones.

1.1.Q8. The set of pages (documents) that contain the term "audit" — so a query for "audit" is a fast lookup, not a scan.

Chapter 2 · robots.txt and Crawl Directives

Parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.

Chapter Skill Map

What this chapter adds: In robots.txt and Crawl Directives, the focus is parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay. You practise it across 40 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples.

chapter arc

What it builds on: Chapter 2 assumes the reading habits from Chapter 1: How Search Engines Work. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 3, XML Sitemaps and URL Discovery, reuses this chapter's discipline with a new kind of problem.

1 **Projects 31-43**	2 **Projects 44-56**	3 **Projects 57-70**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

robots.txt is not a wall — it is a rulebook the crawler reads to itself.

groups, then longest match, then allow-wins ties

Every well-behaved bot, before it fetches anything from `example.com`, requests `https://example.com/robots.txt` and reads it as instructions about where it may go. The file looks like a flat list of lines, but a crawler does not read it flat. It reads it as a set of groups: a `User-agent:` line opens a block, and every `Allow:` / `Disallow:` line beneath it belongs to that block until the next `User-agent:` appears. Parsing the file into those groups is step zero — nothing else works until you have it.

Once the file is grouped, answering a question takes three moves in a fixed order. First, pick the group: a bot obeys *only* the block that names it, and falls back to the `*` block if it has no named block of its own. Second, within that one group, find every rule whose pattern matches the path. Third, apply precedence — the matching rule with the *longest* pattern wins, and if an Allow and a Disallow tie on length, Allow wins. Everything in this chapter is an implementation of those three moves.

The single operation underneath it all is a prefix test: a path is governed by a rule when the path *starts with* the rule's value. `Disallow: /admin` matches `/admin`, `/admin/login`, and `/administrator` alike, because all three begin with `/admin`. Here is that core test, the thing a crawler runs before every fetch:

```
disallow = ['/admin', '/cart']
paths = ['/admin/login', '/blog', '/cart', '/products']
for path in paths:
```

```
    blocked = any(path.startswith(rule) for rule in disallow)
    print(f'{path}: {"blocked" if blocked else "crawl"}')
```

```
output
/admin/login: blocked
/blog: crawl
/cart: blocked
/products: crawl
```

Read that loop and you have read the heart of robots.txt enforcement: a prefix test against a list of forbidden starts. Everything else — Allow overrides, wildcards, per-bot groups — is added precision layered on top of `startswith`.

startswith is the whole engine; the rest is tie-breaking

Two things robots.txt is *not*. It does not hide a page from the index — a disallowed URL can still appear in results if other sites link to it; it only stops the *fetch*. And it is advisory: honest crawlers obey it, but it enforces nothing. Keeping secrets out of robots.txt (which is publicly readable) is itself a classic mistake.

ROBOTS.TXT

A plain-text file at the site root (`/robots.txt`) that tells crawlers which paths they may or may not fetch. It controls **crawling**, not indexing, and is honoured voluntarily.

USER-AGENT GROUP

A `User-agent:` line plus all the `Allow`/`Disallow` lines beneath it, up to the next `User-agent:`. A bot obeys exactly one group: its own named block, or `*` if it has none.

DISALLOW

A rule naming a path prefix the bot must not fetch. `Disallow: /admin` blocks every URL starting with `/admin`. An empty `Disallow:` (no value) blocks nothing — it permits everything.

ALLOW

A rule that carves an exception out of a broader Disallow. `Allow: /blog/public` re-permits a sub-path under `Disallow: /blog` by matching more characters.

LONGEST-MATCH PRECEDENCE

When several rules match a path, the one whose pattern matches the most characters wins; an Allow and a Disallow of equal length are broken in favour of Allow. This — not file order — decides the verdict.

WILDCARD (`*` AND `$`)

In a pattern, ∗ means "any sequence of characters" and a trailing `0` or `/*?sort=`.

Step zero: split the flat file into groups

A crawler cannot reason about robots.txt until it has carved the lines into groups keyed by user-agent. Walk the file top to bottom; a `User-agent:` line opens (or re-opens) a group, and each `Allow`/`Disallow` line is filed under the currently open agent. Split each line on its first colon into a *field* and a *value* — field names are case-insensitive, so lower-case them.

```
robots = """User-agent: Googlebot
Disallow: /no-google

User-agent: *
Disallow: /private
"""
groups, agent = {}, None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    field, _, value = line.partition(':')
    field, value = field.strip().lower(), value.strip()
    if field == 'user-agent':
        agent = value
        groups.setdefault(agent, [])
    elif agent is not None:
        groups[agent].append((field, value))
for ua in sorted(groups):
    print(f'{ua}: {groups[ua]}')
```

```
output
*: [('disallow', '/private')]
Googlebot: [('disallow', '/no-google')]
```

Two subtleties live in this loop. A rule that appears *before* any `User-agent:` belongs to no group and is silently ignored. And if the same bot is named twice, its blocks are merged, not replaced — `setdefault` keeps the existing list and appends to it.

Pick the one group that applies to a bot

A bot does not obey the whole file — it obeys a single group. The selection rule: if the file names the bot, use that block and ignore the * block entirely; otherwise fall back to *; and if there is no * either, the bot has no rules and may crawl everything. Matching is case-insensitive and picks the *most specific* (longest) matching name, so `Googlebot-Image` obeys its own block rather than the generic `Googlebot` one.

```
groups = {'googlebot': ['/a'], 'googlebot-image': ['/b'], '*': ['/c']}
def pick(bot):
    bot = bot.lower()
    best = None
    for ua in groups:
        if ua != '*' and ua in bot:
            if best is None or len(ua) > len(best):
                best = ua
    return best or '*'
for bot in ['Googlebot', 'Googlebot-Image', 'Bingbot']:
    print(f'{bot} -> group {pick(bot)!r}')
```

```
output
Googlebot -> group 'googlebot'
Googlebot-Image -> group 'googlebot-image'
Bingbot -> group '*'
```

`Bingbot` has no block of its own, so it falls through to *. `Googlebot-Image` is the longest name contained in its user-agent string, so it wins over the shorter `googlebot`. *specificity, not file order, chooses the group*

Allow vs Disallow: longest match wins, Allow breaks ties

Inside the chosen group, a path can match several rules at once. The verdict is decided by longest-match precedence: among all matching rules, the one with the longest pattern wins. A short `Disallow` is therefore *punched through* by a longer `Allow` that names a sub-path. Track the best length seen and the action that owns it:

```
rules = [('disallow', '/'), ('allow', '/public'), ('disallow', '/public/secret')]
paths = ['/', '/public', '/public/page', '/public/secret', '/other']
for path in paths:
    best_len, action = -1, 'allow'
    for act, pattern in rules:
        if path.startswith(pattern) and len(pattern) > best_len:
            best_len, action = len(pattern), act
    print(f'{path}: {"crawl" if action == "allow" else "blocked"}')
```

```
output
/: blocked
/public: crawl
/public/page: crawl
```

```
/public/secret: blocked
/other: blocked
```

`Disallow: /` blocks everything, but `Allow: /public` (7 chars) out-matches it for any `/public…` path, and the longer `Disallow: /public/secret` (15 chars) re-blocks just that branch. One subtlety the loop above glosses over: when an Allow and a Disallow match with exactly equal length, the tie goes to **Allow**, so an `Allow: /folder` defeats a same-length `Disallow: /folder`.

Wildcards: translate the pattern to a regex

Real rules use `*` ("any run of characters") and a trailing `$` anchor the end. Then matching is one `re` call.

```
import re
def to_rx(pat):
    rx = ['^']
    for ch in pat:
        rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))
    return re.compile(''.join(rx))
rx = to_rx('/*.pdf$')
for path in ['/report.pdf', '/report.pdf?v=2', '/dir/a.pdf', '/a.pdf.html']:
    print(f'{path}: {"match" if rx.match(path) else "no"}')
```

```
output
/report.pdf: match
/report.pdf?v=2: no
/dir/a.pdf: match
/a.pdf.html: no
```

The `$` stripped out — so `/folder/*.html` (13 effective chars) can out-rank a shorter literal Allow.

Beyond robots.txt: meta robots and X-Robots-Tag

robots.txt governs the *fetch*; a separate family of directives governs whether a fetched page may be *indexed*. An HTML page can carry `<meta name="robots" content="noindex, nofollow">` in its head, and non-HTML files (like PDFs) carry the same controls in an `X-Robots-Tag` HTTP header. When both speak, the most restrictive wins — a header `noindex` overrides a page that says `index`.

```
import re
html = '<meta name="robots" content="index, follow">'
headers = {'X-Robots-Tag': 'noindex'}
def parse(text):
    return {d.strip().lower() for d in text.split(',') if d.strip()}
meta = parse(re.search(r'content="([^"]+)"', html).group(1))
hdr = parse(headers.get('X-Robots-Tag', ''))
combined = meta | hdr
print('meta:', sorted(meta))
```

```
print('header:', sorted(hdr))
print('effective noindex:', 'noindex' in combined)
```

```
output
meta: ['follow', 'index']
header: ['noindex']
effective noindex: True
```

So a URL can be perfectly crawlable in robots.txt yet still kept out of results by a `noindex` — and, conversely, a `Disallow` in robots.txt can stop the bot from ever seeing a `noindex` tag, which is why disallowing a page you want de-indexed is self-defeating.

crawl control and index control are two different switches

Example 1

A robots.txt has `Disallow: /` then `Allow: /public` then `Disallow: /public/secret`, all under `User-agent: *`. For Bingbot, is `/public/page` crawlable, and is `/public/secret`?

SOLUTION

1. Bingbot has no named block, so it falls back to the * group and obeys these three rules.
2. For `/public/page`: it matches `Disallow: /` (1 char) and `Allow: /public` (7 chars). Longest match is the Allow → crawl.
3. For `/public/secret`: it matches `/` (1), `/public` (7) and `Disallow: /public/secret` (15). The longest is the Disallow → blocked.

`/public/page` is crawlable; `/public/secret` is blocked. Longest-match precedence, not file order, decides each one.

more characters matched = higher authority

Example 2

The file has a `Googlebot` block (`Disallow: /no-google`) and a * block (`Disallow: /private`). May Googlebot fetch `/private`? May it fetch `/no-google`?

SOLUTION

1. Googlebot has its own named block, so it obeys *only* that block and ignores the * block entirely.
2. In the Googlebot block the only rule is `Disallow: /no-google`. `/private` does not start with `/no-google`, and no rule blocks it → crawl.
3. `/no-google` does start with `/no-google`, so it is blocked for Googlebot.

Googlebot MAY fetch `/private` (its block never mentions it) and may NOT fetch `/no-google`. The * block's `/private` rule does not apply to a bot that has its own group.

a named bot reads only its own block

Example 3

A rule `Disallow: /*.pdf$` is in force. Which of `/a.pdf`, `/a.pdf?dl=1`, `/dir/b.pdf`, `/a.pdf.html` are blocked?

SOLUTION

1. Compile the pattern: `*` → `.*`, the trailing `0` .
2. `/a.pdf` ends in `.pdf` → match → blocked. `/dir/b.pdf` also ends in `.pdf` → blocked.
3. `/a.pdf?dl=1` continues past `.pdf` with `?dl=1`, so the $ anchor fails → not matched → crawl. `/a.pdf.html` ends in `.html`, not `.pdf` → crawl.

Blocked: `/a.pdf` and `/dir/b.pdf`. Allowed: `/a.pdf?dl=1` and `/a.pdf.html`. The `0` means "and nothing after"))

Mental-Math Shortcut

Group, match, longest wins (Allow on ties). To predict any verdict, run it in that order: pick the one group for the bot (its own name beats *, and a named bot ignores *), find every rule that matches the path as a prefix, then keep the rule with the most matched characters — and if an Allow and a Disallow tie on length, Allow wins. If no rule matches at all, the answer is *crawl*: robots.txt is permissive by default.

⚠ Common Mistakes

- Reading robots.txt as ordered rules where "last wins" or "first wins". Crawlers use longest-match precedence; line order does not decide the verdict.
- Assuming `Disallow` keeps a URL out of Google. It only blocks the fetch — a disallowed URL can still be indexed from external links. Use `noindex` (meta or X-Robots-Tag) to keep a page out of results.
- Confusing `Disallow: /` (blocks the whole site) with `Disallow:` (empty value, allows the whole site). A shipped staging file with `Disallow: /` silently hides everything.
- Disallowing `/css/` or `/js/`, which stops Google from rendering the page as users see it; or putting secret paths in robots.txt, which is publicly readable and just advertises them.

This chapter contains 40 hands-on mini-projects (Projects 31–70). Type each one out, run it, and experiment.

Project 31 Parse robots.txt into User-Agent Rule Groups

The Idea Behind It

Crawlers do not read robots.txt as loose text; they fold it into groups where each User-agent line opens a block owning the Allow and Disallow lines beneath it. This project walks the lines, partitions each on the colon, lowercases the field, and uses setdefault to start a fresh rule list whenever a new agent appears, building the structure every later check depends on.

What You'll Build & Learn

Turn a flat robots.txt string into a dictionary keyed by user-agent, each holding its ordered Allow and Disallow rules.

- robots.txt is grouped by user-agent
- partition splits a field from its value
- setdefault seeds a fresh rule list

🧭 Skill Roadmap

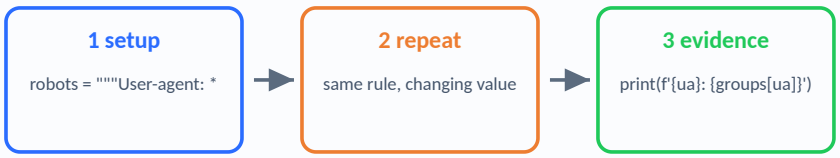
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 32 changes the surface problem while preserving the underlying method. *progress check*

Reading target: parse robots.txt into user-agent rule groups is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{ua}: {groups[ua]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{ua}: {groups[ua]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Parse robots.txt into User-Agent Rule Groups: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, admin. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * | Core: if not line: | Evidence: print(f'{ua}: {groups[ua]}')

🛠️ Reader to Writer Bridge

1. Read: explain parse robots.txt into user-agent rule groups in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 32 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{ua}: {groups[ua]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of parse robots.txt into user-agent rule groups and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Splitting each line on the first colon yields a `field: value` pair; a `user-agent` line opens a new group and following allow/disallow lines append to whichever group is currently open.

```
robots = """User-agent: *
Disallow: /admin
Disallow: /cart
Allow: /
"""
groups = {}
agent = None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    field, _, value = line.partition(':')
    field = field.strip().lower()
    value = value.strip()
    if field == 'user-agent':
        agent = value
        groups.setdefault(agent, [])
    elif field in ('allow', 'disallow') and agent is not None:
        groups[agent].append((field, value))
```

```
    for ua in sorted(groups):
        print(f'{ua}: {groups[ua]}')
```

```
output
*: [('disallow', '/admin'), ('disallow', '/cart'), ('allow', '/')]
```

How it works:

1. A triple-quoted robots string holds one wildcard block; groups starts empty and agent is None.
2. Each line is stripped, blanks are skipped, and partition(':') splits it into field and value.
3. When field equals user-agent the value becomes the current agent and setdefault opens an empty list for it.
4. Allow and Disallow lines append a (field, value) tuple to the active agent's list.
5. Sorting the keys, the final loop prints each agent with its captured rules.

Try it:

1. Add a second User-agent block for Googlebot and confirm both keys appear in the printed dictionary
2. Track the source line number with each rule so you can report where a directive came from
3. Preserve directive order across blocks and print the rules without sorting the agent keys
4. Add a second `User-agent: Googlebot` block and confirm a new key appears.

⚠ Common Mistakes

- Splitting on every colon breaks `Sitemap: https://...` lines whose value itself contains a colon; `partition(':')` splits only on the first.
- Appending allow/disallow rules while `agent` is still None (no User-agent seen yet) silently drops or misattributes them.
- Forgetting to lowercase the field name misses `DISALLOW` or `User-Agent`, since directive names are case-insensitive.

Skills gained string parsing | dict grouping | robots.txt structure

→ Where You'll Use This: Every robots.txt linter, crawler, and SEO audit tool starts by parsing the file into per-agent groups before it can answer any crawlability question.

Mental Model

Think of the file as a filing cabinet: each User-agent line is a drawer label, and the Allow/Disallow lines drop into whichever drawer is currently open.

Project 32 Match a Path Against Disallow Prefixes

The Idea Behind It

The atomic robots.txt operation is a prefix test: a URL path is blocked when it begins with any Disallow value. This project keeps a list of disallowed prefixes and, for each candidate path, uses any() with str.startswith to decide blocked or crawl. It is the exact comparison a crawler runs before deciding whether to fetch a given URL.

What You'll Build & Learn

Classify each path as blocked or crawlable by testing whether it starts with any disallowed prefix.

- a Disallow match is a prefix test
- any() short-circuits over a rule list
- startswith drives the verdict

Skill Roadmap

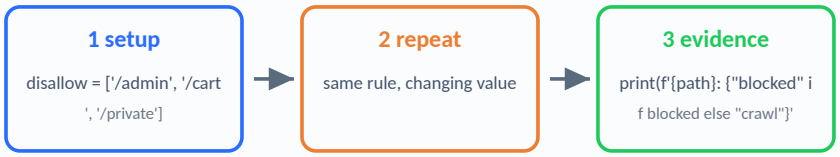
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 31: parse robots.txt into user-agent rule groups. Project 33 changes the surface problem while preserving the underlying method. *progress check*

Reading target: match a path against disallow prefixes is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `disallow = ['/admin', '/cart', '/private']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"blocked" if blocked else "crawl"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`disallow = ['/admin', '/cart', '/private']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`print(f'{path}: {"blocked" if blocked else "crawl"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"blocked" if blocked else "crawl"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{path}: {"blocked" if blocked else "crawl"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Match a Path Against Disallow Prefixes: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: disallow, admin, cart, paths, login. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: disallow = ['/admin', '/cart', '/private'] | Core: print(f'{path}: {"blocked" if blocked else "crawl"}') | Evidence: print(f'{path}: {"blocked" if blocked else "crawl"}')

🛠️ Reader to Writer Bridge

1. Read: explain match a path against disallow prefixes in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 31.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 33 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"blocked" if blocked else "crawl"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `disallow = ['/admin', '/cart', '/private']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of match a path against disallow prefixes and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`str.startswith` over each Disallow value implements robots' prefix matching, so `/admin` blocks `/admin/login` but a path only sharing a prefix-of-a-word like `/privacy` vs `/private` is not blocked.

```
disallow = ['/admin', '/cart', '/private']
paths = ['/admin/login', '/blog', '/cart', '/products', '/privacy']
for path in paths:
    blocked = any(path.startswith(rule) for rule in disallow)
    print(f'{path}: {"blocked" if blocked else "crawl"}')
```

```
output
/admin/login: blocked
/blog: crawl
/cart: blocked
```

```
/products: crawl
/privacy: crawl
```

How it works:

1. disallow lists three blocked prefixes and paths lists five candidate URLs.
2. For each path, any(path.startswith(rule) for rule in disallow) returns True if a prefix matches.
3. The conditional expression prints blocked when the generator finds a match and crawl otherwise.
4. /admin/login and /cart match prefixes and print blocked; /privacy does not match /private so it crawls.

Try it:

1. Add /search to the disallow list and verify a /search?q=x style path becomes blocked
2. Return the specific rule that matched instead of a plain blocked label for easier debugging
3. Make /privacy correctly crawlable even if you add a /priv prefix by testing for a path separator boundary
4. Add `/cartoon` to the paths and confirm `/cart` still blocks it (prefix match).

⚠ Common Mistakes

- Assuming `/private` blocks `/privacy`; it does not, because `/privacy` does not start with the full string `/private` (they differ at the 8th character).
- Using `in` instead of `startswith` would wrongly block any path that merely contains the rule text anywhere.
- Forgetting that an empty Disallow value matches every path because every string starts with the empty string.

Skills gained prefix matching | startswith | any()

→ Where You'll Use This: Prefix testing is the inner loop of any crawl-budget audit that flags which of a site's URLs a robots.txt would keep search engines from fetching.

Mental Model

It works like a bouncer scanning the start of each name on a guest list: if the path's opening characters match a banned prefix, the door stays shut.

Project 33 Let Allow Override a Broader Disallow

The Idea Behind It

Allow and Disallow lines live together, and conflicts resolve by specificity: the longer matching rule wins. This project computes, per path, the longest Allow prefix length and the longest Disallow prefix length using max with a default, then crawls when the Allow match is at least as long. A narrow Allow can therefore punch a hole through a broad Disallow.

What You'll Build & Learn

Let a longer Allow prefix override a shorter Disallow so a permitted sub-path stays crawlable inside a blocked folder.

- longer matching rule wins
- max with default guards empty matches
- Allow can override Disallow

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 32: match a path against disallow prefixes. Project 34 changes the surface problem while preserving the underlying method.

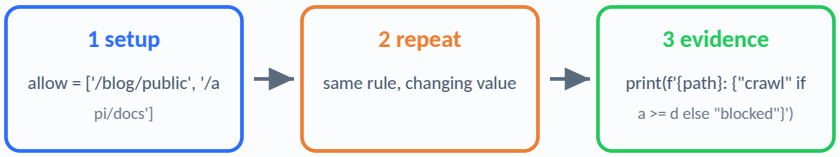
progress check

Reading target: let allow override a broader disallow is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `allow = ['/blog/public', '/api/docs']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"crawl" if a >= d else "blocked"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`allow = ['/blog/public', '/api/docs']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`a = max((len(r) for r in allow if path.startswith(r)), default=-1)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"crawl" if a >= d else "blocked"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `a = max((len(r) for r in allow if path.startswith(r)), default=-1)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Let Allow Override a Broader Disallow: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: allow, blog, api, docs, disallow. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: allow = ['/blog/public', '/api/docs'] | Core: a = max *len(r) for r in allow if path.startswith(r*, default=-1) | Evidence: print(f'{path}: {"crawl" if a >= d else "blocked"}')

🛠️ Reader to Writer Bridge

1. Read: explain let allow override a broader disallow in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 32.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 34 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"crawl" if a >= d else "blocked"}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `allow = ['/blog/public', '/api/docs']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of let allow override a broader disallow and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

For each path the longest matching Allow length is compared to the longest matching Disallow length; the longer match decides, so a specific `/blog/public` Allow beats a broad `/blog` Disallow.

```
allow = ['/blog/public', '/api/docs']
disallow = ['/blog', '/api']
paths = ['/blog/public/post', '/blog/draft', '/api/docs', '/api/keys']
for path in paths:
    a = max((len(r) for r in allow if path.startswith(r)), default=-1)
    d = max((len(r) for r in disallow if path.startswith(r)), default=-1)
    print(f'{path}: {"crawl" if a >= d else "blocked"}')
```

```
output
/blog/public/post: crawl
/blog/draft: blocked
/api/docs: crawl
/api/keys: blocked
```

How it works:

1. allow holds two specific sub-paths and disallow holds two broad parent paths.

2. For each path, a is the longest matching Allow length and d the longest matching Disallow length, defaulting to -1.
3. The ternary prints crawl when a >= d, meaning the Allow match is at least as specific.
4. /blog/public/post crawls because /blog/public is longer than /blog; /blog/draft stays blocked.

Try it:

1. Add an Allow for /api/keys/help and check it overrides the broader /api Disallow
2. Print both the winning length and the action so a tie at -1 versus -1 is visible
3. Change the comparison to strict greater-than and observe which equal-length paths flip to blocked
4. Add `('Allow', '/api/keys/public')` and watch that one sub-path open up.

⚠ Common Mistakes

- Using `>` instead of `>=` flips the tie toward Disallow; Google's rule is that Allow wins ties of equal length.
- Forgetting the `default=-1` makes `max` raise on a path that matches no rule of one kind.
- Comparing the number of matching rules instead of the matched prefix length gives the wrong winner.

Skills gained longest-match | max default | rule precedence

→ Where You'll Use This: Specificity resolution is what lets SEO teams open a single public folder inside an otherwise disallowed section without rewriting the whole robots.txt.

Mental Model

Picture two highlighters over the same URL: the one that colours more characters of the path is the one whose verdict sticks.

Project 34 Select the Rule Group for a Named Bot

The Idea Behind It

When robots.txt holds both a named Googlebot block and a wildcard block, Googlebot follows only its own block and ignores the star entirely. This project parses every group, then selects the bot's block if it exists and falls back to the star otherwise, printing the chosen group and its rules. Picking the correct group is what makes per-bot targeting actually work.

What You'll Build & Learn

Choose the rule group a named bot must obey, preferring its own block over the wildcard fallback.

- a named bot ignores the star block
- group selection precedes rule evaluation

- fallback to wildcard when unnamed

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 33: let allow override a broader disallow. Project 35 changes the surface problem while preserving the underlying method.

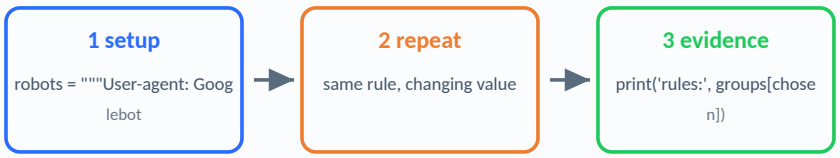
progress check

Reading target: select the rule group for a named bot is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `robots = """User-agent: Googlebot` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print('rules:', groups[chosen])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: Googlebot`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print('rules:', groups[chosen])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Select the Rule Group for a Named Bot: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Googlebot, Disallow. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: Googlebot | Core: if not line: | Evidence: print('rules:', groups[chosen])

Reader to Writer Bridge

1. Read: explain select the rule group for a named bot in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 33.
   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 35 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('rules:', groups[chosen])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: Googlebot` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of select the rule group for a named bot and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A bot uses its most specific matching group; if a group names the bot exactly it wins outright, and the `*` group is used only as a fallback when no named group matches.

```
robots = """User-agent: Googlebot
Disallow: /no-google

User-agent: *
Disallow: /private
"""
groups, agent = {}, None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    field, _, value = line.partition(':')
    field, value = field.strip().lower(), value.strip()
    if field == 'user-agent':
        agent = value
        groups.setdefault(agent, [])
    elif agent is not None:
        groups[agent].append((field, value))
bot = 'Googlebot'
chosen = bot if bot in groups else '*'
print(f'group for {bot}: {chosen}')
print('rules:', groups[chosen])
```

```
output
group for Googlebot: Googlebot
rules: [('disallow', '/no-google')]
```

How it works:

1. The robots string defines a Googlebot block and a separate wildcard block, separated by a blank line.
2. The parser groups rules per agent exactly as before, skipping blanks and lowercasing fields.
3. chosen is set to bot when 'Googlebot' is a key, else to the wildcard '*'.
4. Because Googlebot has its own block, it prints that group's single /no-google rule, not the wildcard's /private.

Try it:

1. Change bot to Bingbot and confirm it falls back to the wildcard group's rules
2. Add a /no-google/exception Allow to Googlebot's block and reprint the chosen rules
3. Report which group was used so a fallback to the star is explicit in the output
4. Set `bot = 'Bingbot'` and watch it fall back to the `*` group.

⚠ Common Mistakes

- Merging the named group with the `*` group doubles up rules; a bot must obey one group only.
- Falling back to `*` when a named group exists but is empty wrongly applies the wildcard rules.
- Treating the blank line between groups as optional; it is what separates one User-agent block from the next.

Skills gained group selection | wildcard fallback | dict lookup

→ Where You'll Use This: Correct group selection underpins any tool that answers crawlability per crawler, since Googlebot, Bingbot, and AdsBot each obey only their own most-specific block.

Mental Model

It is like a memo addressed to one person: if your name is on the envelope you read only that copy and disregard the all-staff notice pinned to the wall.

Project 35 Resolve Verdicts by Longest-Match Precedence

The Idea Behind It

With overlapping Allow and Disallow lines, the rule whose pattern matches the most characters decides the verdict. This project scans a flat rule list, tracking the longest prefix that the path starts with and remembering its action, then translates allow to crawl and disallow to blocked. This longest-match precedence is the core of how Google evaluates robots.txt.

What You'll Build & Learn

Resolve each path to crawl or blocked by keeping the action of the longest matching rule pattern.

- longest pattern sets the verdict
- track best length while scanning
- Google uses longest-match

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 34: select the rule group for a named bot. Project 36 changes the surface problem while preserving the underlying method.

progress check

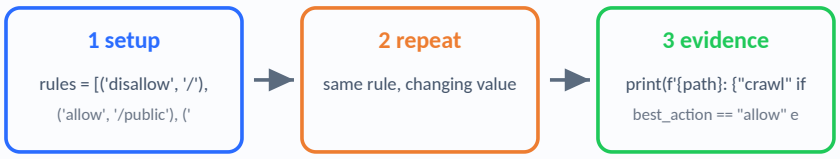
Reading target: resolve verdicts by longest-match precedence is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `rules = [('disallow', '/'), ('allow', '/public'), ('disallow', '/public/secret')]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"crawl" if best_action == "allow" else "blocked"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`rules = [('disallow', '/'), ('allow', '/public'), ('disallow', '/public/secret')]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse

Move	Code to inspect	Why it matters
		and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if path.startswith(pattern) and len(pattern) > best_len:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"crawl" if best_action == "allow" else "blocked"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if path.startswith(pattern) and len(pattern) > best_len:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Resolve Verdicts by Longest-Match Precedence: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: rules, disallow, allow, secret, paths. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: rules = [('disallow', '/'), ('allow', '/public'), ('disallow', '/public... | Core: if path.startswith(pattern) and len(pattern) > best_len: | Evidence: print(f'{path}: {"crawl" if best_action == "allow" else "blocked"}')

Reader to Writer Bridge

1. Read: explain resolve verdicts by longest-match precedence in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 34.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 36 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"crawl" if best_action == "allow" else "blocked"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `rules = [('disallow', '/'), ('allow', '/public'), ('disallow', '/public/secret')]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of resolve verdicts by longest-match precedence and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Scanning every rule and keeping the action of the longest matching prefix means a deep `/public/secret` Disallow overrides a shorter `/public` Allow, while `/other` matches only the site-wide `/` Disallow and stays blocked.

```
rules = [('disallow', '/'), ('allow', '/public'), ('disallow',
'/public/secret')]
paths = ['/', '/public', '/public/page', '/public/secret', '/other']
for path in paths:
    best_len, best_action = -1, 'allow'
    for action, pattern in rules:
        if path.startswith(pattern) and len(pattern) > best_len:
            best_len, best_action = len(pattern), action
    print(f'{path}: {"crawl" if best_action == "allow" else "blocked"}')
```

```
output
/: blocked
/public: crawl
/public/page: crawl
/public/secret: blocked
/other: blocked
```

How it works:

1. rules holds a root Disallow, a /public Allow, and a deeper /public/secret Disallow.
2. For each path, best_len starts at -1 and best_action defaults to allow.
3. Every rule whose pattern the path starts with, and that is longer than best_len, updates both trackers.
4. /public/secret is blocked by the 14-character rule, while /public crawls thanks to its 7-character Allow.

Try it:

1. Add an Allow for /public/secret/ok and confirm the deepest rule now permits that path
2. Print the winning pattern alongside the verdict to make the longest match visible
3. Reverse the rule list order and verify the result is unchanged because length, not order, decides
4. Add `('allow', '/public/secret/ok')` and confirm that deepest path opens again.

⚠ Common Mistakes

- First-match-wins instead of longest-match gives the wrong answer the moment two rules overlap.
- Initialising `best_action` to 'disallow' makes unmatched paths blocked instead of the default allow.
- Comparing `len(pattern)` without the `startswith` guard counts non-matching rules.

Skills gained longest-match | precedence loop | verdict mapping

→ Where You'll Use This: Longest-match resolution is the engine behind robots.txt testers, so an SEO can predict exactly how Google ranks competing Allow and Disallow lines on a single URL.

⚡ Mental Model

Think of competing claims on a plot of land: the surveyor sides with whoever's boundary line covers the most ground.

Project 36 Break a Specificity Tie in Allow's Favour

💡 The Idea Behind It

When an Allow and a Disallow match a path with identical pattern lengths, the tie goes to Allow. This project tracks the longest match but, on an exact length tie, flips allow_wins to True whenever the tying rule is an Allow. That single tie-break keeps a hesitant Disallow: /folder from blocking a path that an equally specific Allow: /folder was meant to open.

What You'll Build & Learn

Break an exact-length tie between Allow and Disallow in favour of Allow so the path stays crawlable.

- equal-length ties favour Allow
- track allow_wins alongside length

- tie-break overrides Disallow

Skill Roadmap

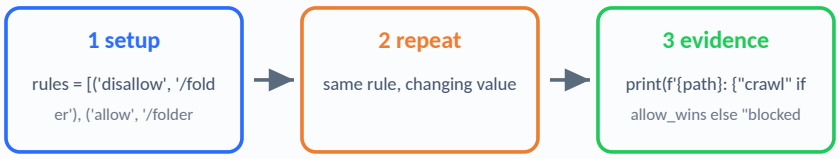
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 35: resolve verdicts by longest-match precedence. Project 37 changes the surface problem while preserving the underlying method. *progress check*

Reading target: break a specificity tie in allow's favour is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `rules = [('disallow', '/folder'), ('allow', '/folder')]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"crawl" if allow_wins else "blocked"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`rules = [('disallow', '/folder'), ('allow', '/folder')]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if path.startswith(pattern):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'{path}: {"crawl" if allow_wins else "blocked"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if path.startswith(pattern):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Break a Specificity Tie in Allow's Favour: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: rules, disallow, folder, allow, paths. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: rules = [('disallow', '/folder'), ('allow', '/folder')] | Core: if path.startswith(pattern): | Evidence: print(f'{path}: {"crawl" if allow_wins else "blocked"}')

Reader to Writer Bridge

1. Read: explain break a specificity tie in allow's favour in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 35.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 37 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"crawl" if allow_wins else "blocked"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `rules = [('disallow', '/folder'), ('allow', '/folder')]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of break a specificity tie in allow's favour and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

When a new match equals the current best length, the code only flips the verdict if the new rule is an Allow, encoding Google's 'Allow wins equal-length ties' rule.

```
rules = [('disallow', '/folder'), ('allow', '/folder')]
paths = ['/folder', '/folder/page', '/other']
for path in paths:
    best_len, allow_wins = -1, True
    for action, pattern in rules:
        if path.startswith(pattern):
            if len(pattern) > best_len:
                best_len, allow_wins = len(pattern), (action == 'allow')
            elif len(pattern) == best_len and action == 'allow':
                allow_wins = True
    print(f'{path}: {"crawl" if allow_wins else "blocked"}')
```

```
output
/folder: crawl
/folder/page: crawl
/other: crawl
```

How it works:

1. rules pairs a Disallow and an Allow on the identical pattern /folder.
2. For each path, best_len starts at -1 and allow_wins starts True.
3. A strictly longer match resets allow_wins to whether that rule is an Allow.
4. On an equal-length match, allow_wins flips to True only when the tying rule is an Allow, so every path crawls.

Try it:

1. Make the Disallow pattern one character longer and watch the verdict flip to blocked
2. Add a print of allow_wins per rule iteration to trace how the tie resolves
3. Introduce a third rule at the same length and confirm an Allow still wins the group
4. Swap the two rules' order and confirm the verdict does not change.

⚠ Common Mistakes

- Letting the last-seen rule win the tie makes the order of lines change the verdict.
- Initialising `allow_wins` to False would block `/other`, which no rule even matches.
- Forgetting the equal-length branch makes Disallow win ties whenever it is listed first.

Skills gained tie-breaking | Allow precedence | match tracking

→ Where You'll Use This: The equal-length Allow-wins rule matters whenever a config has mirror-image directives, and getting it right prevents an audit from falsely reporting a permitted folder as blocked.

⚡ Mental Model

Like a referee's call on an even play, the benefit of the doubt goes to the side that lets the action continue, so Allow carries a tied decision.

Project 37 Convert a Wildcard Pattern to a Regex

💡 The Idea Behind It

robots.txt patterns let an asterisk stand for any run of characters, which is more than plain prefix matching can express. This project compiles a pattern into a regex by escaping every literal character with re.escape and turning each star into .*, then anchors it with a leading caret. Matching a path against that compiled regex brings full wildcard support using only the standard library.

What You'll Build & Learn

Translate a robots wildcard pattern into an anchored regex so star segments match any characters.

- star means any character sequence
- re.escape protects literals
- anchor patterns with a caret

🧭 Skill Roadmap

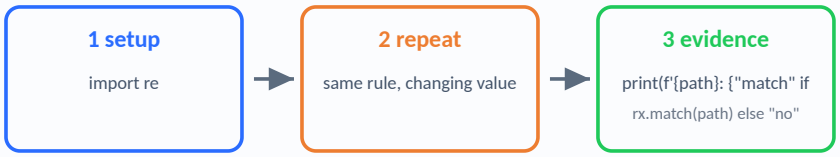
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 36: break a specificity tie in allow's favour. Project 38 changes the surface problem while preserving the underlying method. *progress check*

Reading target: convert a wildcard pattern to a regex is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"match" if rx.match(path) else "no"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if ch == '*':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"match" if rx.match(path) else "no"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if ch == '*':`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Convert a Wildcard Pattern to a Regex: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, to_regex, pattern, ch, append. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if ch == '*': | Evidence: print(f'{path}: {"match" if rx.match(path) else "no"}')

Reader to Writer Bridge

1. Read: explain convert a wildcard pattern to a regex in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 36.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 38 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"match" if rx.match(path) else "no"}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of convert a wildcard pattern to a regex and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Building the regex character by character, escaping every literal and expanding `*` to `.*`, turns a robots pattern like `/*.php` into a safe, anchored regular expression.

```
import re
def to_regex(pattern):
    out = ['^']
    for ch in pattern:
        if ch == '*':
            out.append('.*')
        else:
            out.append(re.escape(ch))
    return ''.join(out)
pattern = '/*.php'
rx = re.compile(to_regex(pattern))
paths = ['/index.php', '/dir/page.php', '/about.html', '/x.php?id=1']
for path in paths:
    print(f'{path}: {"match" if rx.match(path) else "no"}')
```

```
output
/index.php: match
/dir/page.php: match
/about.html: no
/x.php?id=1: match
```

How it works:

1. to_regex builds a list starting with '^' and walks each character of the pattern.
2. A star appends '.*' while any other character is passed through re.escape and appended.
3. The joined string compiles into rx, and */.php becomes ^/.\.php.*
4. /index.php and /x.php?id=1 match because .* absorbs the leading and trailing text; /about.html does not.

Try it:

1. Add a pattern /admin/* and confirm it matches nested admin paths but not /administrator

2. Print the generated regex string to inspect how each character was escaped
3. Use fullmatch instead of match and observe which trailing-text paths stop matching
4. Change the pattern to `/admin*` and test it against `/administrator`.

⚠ Common Mistakes

- Skipping `re.escape` lets pattern characters like `.` or `?` act as regex metacharacters and over-match.
- Forgetting the leading `^` anchor lets the pattern match anywhere in the path, not from the start.
- Expecting `/*.php` to require a directory; the leading `*` can match the empty string, so `/index.php` matches too.

Skills gained wildcard to regex | re.escape | pattern anchoring

→ Where You'll Use This: Wildcard translation is essential for any robots checker that must honour patterns like */.php or /?*, which appear constantly in real e-commerce and CMS robots files.

Mental Model

The asterisk is a blank tile in a word game: it fills in for any letters, and re.escape keeps the real letters from being misread as game rules.

Project 38 Anchor a Pattern to the Path End with $

The Idea Behind It

A trailing dollar sign in a robots pattern means the path must end exactly there. This project extends the wildcard-to-regex translator to map a literal $ onto a regex end-anchor, so /*.pdf$ matches a bare PDF but rejects one carrying a query string. It is precisely how you block .pdf files while letting .pdf?download=1 URLs slip through.

What You'll Build & Learn

Add end-anchoring so a pattern can match a file extension exactly and reject anything appended after it.

- trailing $ anchors the path end
- map $ to a regex end-anchor
- queries defeat an exact-end match

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 37: convert a wildcard pattern to a regex. Project 39 changes the surface problem while preserving the underlying method.

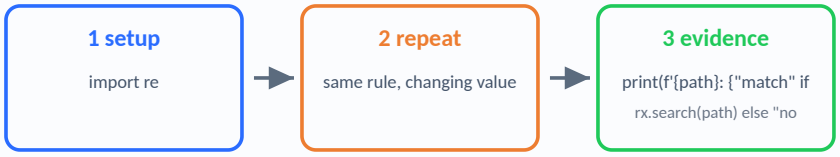
progress check

Reading target: anchor a pattern to the path end with $ is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"match" if rx.search(path) else "no"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if ch == '*':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"match" if rx.search(path) else "no"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if ch == '*':`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Anchor a Pattern to the Path End with $: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, to_regex, pattern, ch, append. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if ch == '*': | Evidence: print(f'{path}: {"match" if rx.search(path) else "no"}')

🛠️ Reader to Writer Bridge

1. Read: explain anchor a pattern to the path end with $ in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 37.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 39 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"match" if rx.search(path) else "no"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of anchor a pattern to the path end with $ and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Mapping `$` to the regex end-anchor `$` means `/*.pdf$` matches a path ending in `.pdf` but not one with a query string after it, since the characters after `.pdf` break the anchor.

```
import re
def to_regex(pattern):
    out = ['^']
    for ch in pattern:
        if ch == '*':
            out.append('.*')
        elif ch == '$':
            out.append('$')
        else:
            out.append(re.escape(ch))
    return ''.join(out)
pattern = '/*.pdf$'
rx = re.compile(to_regex(pattern))
paths = ['/report.pdf', '/report.pdf?v=2', '/dir/a.pdf', '/a.pdf.html']
for path in paths:
    print(f'{path}: {"match" if rx.search(path) else "no"}')
```

```
output
/report.pdf: match
/report.pdf?v=2: no
/dir/a.pdf: match
/a.pdf.html: no
```

How it works:

1. to_regex now has a branch that appends a regex '$' when it meets a literal $ in the pattern.
2. */.pdf$ compiles to* ^/.\.pdf$, demanding the path finish at .pdf.
3. Using rx.search, /report.pdf matches but /report.pdf?v=2 fails because text follows .pdf.
4. /a.pdf.html also fails since the path does not end at .pdf despite containing it.

Try it:

1. Add a /*.json$ pattern and verify /data.json blocks while /data.json?cb=1 does not
2. Switch rx.search back to rx.match and check whether mid-path matches still behave
3. Allow a single optional trailing slash before the end-anchor and retest the PDF cases

4. Remove the `$` from the pattern and watch `/report.pdf?v=2` start matching.

⚠ Common Mistakes

- Treating `$` as a literal dollar sign blocks the wrong URLs and never anchors.
- Forgetting that the query string `?v=2` is part of the path string here, so `$` correctly rejects it.
- Using `rx.match` here still works, but dropping the `^` from the builder would let `.pdf` match mid-path.

Skills gained end anchoring | regex $ | extension matching

→ Where You'll Use This: End-anchored patterns are how sites block indexable file types like PDFs or .json exactly, a frequent directive in robots files protecting downloads and APIs.

⚡ Mental Model

The dollar sign is a full stop on the URL sentence: the path has to finish right there, so anything written after it breaks the match.

Project 39 Extract Sitemap Lines from robots.txt

💡 The Idea Behind It

Sitemap directives stand outside any user-agent group and point crawlers at the site's URL inventory. This project ignores group structure entirely, scanning every line, partitioning on the colon, and collecting the value whenever the field is sitemap. It prints the count and each URL, mirroring the first thing a crawler does after fetching robots.txt: harvest the sitemaps.

What You'll Build & Learn

Pull every Sitemap URL out of robots.txt regardless of which user-agent group surrounds it.

- Sitemap lines are group-independent
- scan all lines for one field
- collect values into a list

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 38: anchor a pattern to the path end with $. Project 40 changes the surface problem while preserving the underlying method. *progress check*

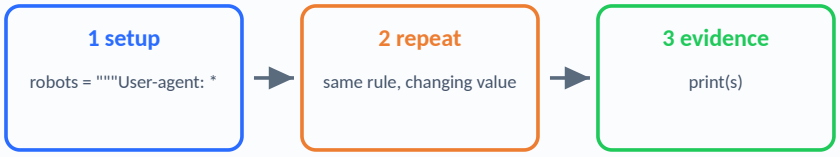
Reading target: extract sitemap lines from robots.txt is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(s)` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if field.strip().lower() == 'sitemap':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(s)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if field.strip().lower() == 'sitemap':`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Extract Sitemap Lines from robots.txt: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, admin. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * | Core: if field.strip().lower() == 'sitemap': | Evidence: print(s)

🛠️ Reader to Writer Bridge

1. Read: explain extract sitemap lines from robots.txt in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 38.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 40 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(s)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of extract sitemap lines from robots.txt and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A `Sitemap` line is global, so it is collected regardless of the current group; `partition(':')` keeps the `https://` URL intact because it splits only on the first colon.

```
robots = """User-agent: *
Disallow: /admin
Sitemap: https://example.com/sitemap.xml
Sitemap: https://example.com/news-sitemap.xml
"""
sitemaps = []
for line in robots.splitlines():
    field, _, value = line.partition(':')
    if field.strip().lower() == 'sitemap':
        sitemaps.append(value.strip())
print('found', len(sitemaps), 'sitemaps')
for s in sitemaps:
    print(s)
```

```
output
found 2 sitemaps
https://example.com/sitemap.xml
https://example.com/news-sitemap.xml
```

How it works:

1. The robots string mixes a wildcard block with two Sitemap lines.
2. Each line is partitioned on ':' and its field is stripped and lowercased.
3. When the field equals sitemap, the cleaned value is appended to the sitemaps list.
4. The code prints found 2 sitemaps, then each captured sitemap URL on its own line.

Try it:

1. Add a third Sitemap line and confirm the printed count updates to three
2. Validate that each captured value starts with https before adding it to the list
3. Deduplicate the sitemaps list so a repeated URL is only reported once
4. Add a third Sitemap line and confirm the count rises.

⚠ Common Mistakes

- Splitting on all colons truncates the URL right after `https`; partition on the first colon avoids that.

- Scoping sitemaps to the current user-agent group; sitemap directives apply site-wide, not per-bot.
- Forgetting to `.strip()` the value leaves a leading space on every captured URL.

Skills gained sitemap extraction | line scanning | field filtering

→ Where You'll Use This: Harvesting Sitemap directives is step one of crawl discovery, feeding the URL set that sitemap validators and index-coverage audits then crawl and verify.

Mental Model

Sitemap lines are the address book stapled to the front of the rulebook: whoever reads the file copies down those addresses before doing anything else.

Project 40 Read Crawl-Delay into a Request Budget

The Idea Behind It

Crawl-delay asks bots to pause a set number of seconds between fetches. This project scans for the crawl-delay field, parses its integer value, and when present converts it into a per-minute fetch ceiling with 60 divided by the delay. Turning the raw directive into a concrete fetch budget makes the politeness rule actionable for a scheduler instead of a number on a page.

What You'll Build & Learn

Read a Crawl-delay value and convert it into a maximum fetches-per-minute politeness budget.

- Crawl-delay sets seconds between fetches
- absent delay means crawl freely
- convert delay to a rate ceiling

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 39: extract sitemap lines from robots.txt. Project 41 changes the surface problem while preserving the underlying method. *progress check*

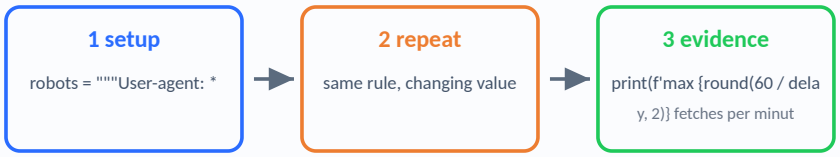
Reading target: read crawl-delay into a request budget is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'max {round(60 / delay, 2)} fetches per minute')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if field.strip().lower() == 'crawl-delay':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'max {round(60 / delay, 2)} fetches per minute')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if field.strip().lower() == 'crawl-delay':`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Read Crawl-Delay into a Request Budget: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Crawl, delay. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * | Core: if field.strip().lower() == 'crawl-delay': | Evidence: print(f'max {round(60 / delay, 2)} fetches per minute')

🛠️ Reader to Writer Bridge

1. Read: explain read crawl-delay into a request budget in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 39.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 41 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'max {round(60 / delay, 2)} fetches per minute')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of read crawl-delay into a request budget and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Parsing the integer after `Crawl-delay:` gives the inter-fetch gap, and `60 / delay` converts that gap into the maximum number of fetches a polite crawler makes per minute.

```
robots = """User-agent: *
Crawl-delay: 10
Disallow: /tmp
"""
delay = None
for line in robots.splitlines():
    field, _, value = line.partition(':')
    if field.strip().lower() == 'crawl-delay':
        delay = int(value.strip())
if delay is None:
    print('no crawl-delay set (crawl freely)')
else:
    print(f'wait {delay}s between fetches')
    print(f'max {round(60 / delay, 2)} fetches per minute')
```

```
output
wait 10s between fetches
max 6.0 fetches per minute
```

How it works:

1. delay starts as None and each line is partitioned and checked for the crawl-delay field.
2. A matching line parses its value with int into delay.
3. When delay is None the code prints a crawl-freely message.
4. Otherwise it prints the wait time and round(60 / delay, 2), giving 6.0 fetches per minute for a 10-second delay.

Try it:

1. Change the delay to 5 and verify the ceiling rises to twelve fetches per minute
2. Guard against a zero delay so the division does not raise an error
3. Accept a fractional delay like 0.5 by parsing with float instead of int
4. Change the delay to 5 and watch the per-minute ceiling double.

⚠ Common Mistakes

- Leaving the value as a string makes `60 / delay` raise a TypeError.
- Dividing by a zero crawl-delay raises ZeroDivisionError; real parsers treat 0 as no delay.
- Assuming every crawler honours Crawl-delay; Googlebot ignores it and uses its own rate logic.

Skills gained crawl-delay parsing | rate conversion | int casting

→ Where You'll Use This: Crawl-delay budgeting feeds polite crawler schedulers and helps SEOs reason about how slowly a directive forces a bot to traverse a large site.

⚡ Mental Model

Crawl-delay is a turnstile timer: it spaces visitors out by N seconds, and dividing a minute by that spacing tells you how many can pass through.

Project 41 Detect a Site-Wide Disallow Block

💡 The Idea Behind It

A Disallow: / blocks the entire site for the matched user-agent, because every path begins with a slash. This project gathers all Disallow values and tests whether a bare slash is among them, flagging a full-site block. Spotting it warns you that a deploy may have shipped a staging robots.txt that quietly hides every page from search engines.

What You'll Build & Learn

Detect a site-wide block by checking whether a lone slash appears among the collected Disallow values.

- Disallow: / blocks the whole site
- every path starts with a slash
- watch for leaked staging rules

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 40: read crawl-delay into a request budget. Project 42 changes the surface problem while preserving the underlying method.

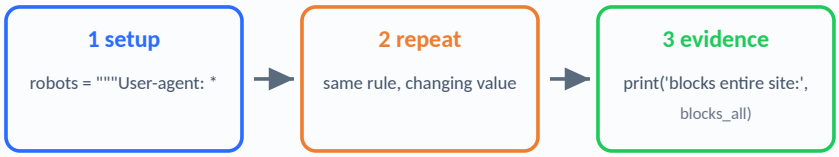
progress check

Reading target: detect a site-wide disallow block is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print('blocks entire site:', blocks_all)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if field.strip().lower() == 'disallow':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('blocks entire site:', blocks_all)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if field.strip().lower() == 'disallow':`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Site-Wide Disallow Block: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, disallows. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * | Core: if field.strip().lower() == 'disallow': | Evidence: print('blocks entire site:', blocks_all)

🛠️ Reader to Writer Bridge

1. Read: explain detect a site-wide disallow block in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 40.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 42 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('blocks entire site:', blocks_all)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. `Transfer:` change one small part of detect a site-wide disallow block and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Because every URL path begins with `/`, a single `Disallow: /` rule matches everything; detecting that exact value flags a total block.

```
robots = """User-agent: *
Disallow: /
"""
disallows = []
for line in robots.splitlines():
    field, _, value = line.partition(':')
    if field.strip().lower() == 'disallow':
        disallows.append(value.strip())
blocks_all = '/' in disallows
print('disallow rules:', disallows)
print('blocks entire site:', blocks_all)
```

```
output
disallow rules: ['/']
blocks entire site: True
```

How it works:

1. The robots string contains a single Disallow: / under the wildcard agent.
2. Each line is partitioned, and disallow values are appended into the disallows list.
3. blocks_all is True when '/' is a member of that list.
4. The code prints the disallow rules and the boolean confirming the entire site is blocked.

Try it:

1. Add a real Allow: / alongside the Disallow and decide how your check should treat the conflict
2. Exit with a nonzero status code when a full-site block is detected for CI use
3. Report which user-agent owns the blanket Disallow rather than just a boolean
4. Change the rule to `Disallow: /admin` and confirm it no longer blocks the whole site.

⚠ Common Mistakes

- Confusing `Disallow: /` (block all) with `Disallow:` empty (allow all); the slash is the only difference.
- Checking `'/' in line` instead of the parsed value matches any rule containing a slash.
- Assuming `Disallow: /*` is needed for a full block; the bare `/` already blocks everything.

Skills gained site-wide block detection membership test value collection

→ Where You'll Use This: A lone Disallow: / is the single most damaging robots mistake, so deploy checks and SEO monitors specifically watch for it to prevent a site vanishing from search.

⚡ Mental Model

Disallow: / is pulling the master breaker: one line cuts power to every room, which is exactly why a leaked staging file is so dangerous.

Project 42 Treat an Empty Disallow as Allow-All

💡 The Idea Behind It

A Disallow with no value is the mirror opposite of Disallow: /: it explicitly opens the whole site to a bot. This project collects Disallow values and, when every entry is an empty string, reports that the entire site is allowed; otherwise it lists the real blocked prefixes. The empty value is a common, deliberate way to grant a crawler unrestricted access.

What You'll Build & Learn

Recognise that an empty Disallow value permits the whole site rather than blocking anything.

- empty Disallow means allow everything
- it is the opposite of Disallow: /
- filter out blank values

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 41: detect a site-wide disallow block. Project 43 changes the surface problem while preserving the underlying method.

progress check

Reading target: treat an empty disallow as allow-all is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.

- Evidence: `print('blocked prefixes:', [r for r in rules if r])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if field.strip().lower() == 'disallow':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('blocked prefixes:', [r for r in rules if r])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if field.strip().lower() == 'disallow':`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Treat an Empty Disallow as Allow-All: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, rules. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * | Core: if field.strip().lower() == 'disallow': | Evidence: print('blocked prefixes:', [r for r in rules if r])

🛠️ Reader to Writer Bridge

1. Read: explain treat an empty disallow as allow-all in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 41.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 43 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('blocked prefixes:', [r for r in rules if r])`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of treat an empty disallow as allow-all and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

An empty Disallow value matches nothing, so a group whose only rule is `Disallow:` imposes no restriction at all, the canonical allow-everything signal.

```
robots = """User-agent: *
Disallow:
"""
rules = []
for line in robots.splitlines():
    field, _, value = line.partition(':')
    if field.strip().lower() == 'disallow':
        rules.append(value.strip())
if rules == [''] or all(r == '' for r in rules):
    print('empty Disallow -> entire site allowed')
else:
    print('blocked prefixes:', [r for r in rules if r])
```

```
output
empty Disallow -> entire site allowed
```

How it works:

1. The robots string has a Disallow line with nothing after the colon.
2. Each Disallow value is stripped and appended to rules, so the empty line yields an empty string.
3. The check rules == [''] or all values being empty triggers the allow-all message.
4. Otherwise the code would print the non-empty blocked prefixes instead.

Try it:

1. Add a real Disallow: /admin and confirm the code now lists blocked prefixes
2. Mix an empty and a non-empty Disallow and decide which message should win
3. Print a count of empty versus non-empty Disallow lines for clarity
4. Add a non-empty `Disallow: /admin` line and watch the verdict change.

⚠ Common Mistakes

- Treating an empty Disallow as a blank line and dropping it loses the explicit allow-all intent.
- Comparing the value to `'/'` mistakes allow-all for block-all.

- Forgetting that an empty Disallow next to real Disallow lines does not undo them.

Skills gained empty-value handling | allow-all detection | list filtering

→ Where You'll Use This: Distinguishing an empty Disallow from a missing one keeps an audit from misreporting a deliberately open robots block as either broken or restrictive.

Mental Model

An empty Disallow is an open gate with the sign still hanging: the gate names a rule but lets everyone walk straight through.

Project 43 Strip Comments Before Parsing Rules

The Idea Behind It

A hash starts a comment that runs to the end of its line, whether the line is all comment or a directive with a trailing note. This project splits each line on its first hash, keeps the part before it, strips whitespace, and discards lines that become empty. Cleaning comments first stops directive parsing from choking on human annotations mixed into the file.

What You'll Build & Learn

Strip whole-line and trailing comments so only clean directive text survives for later parsing.

- a hash begins a comment
- comments run to line end
- clean before parsing directives

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 42: treat an empty disallow as allow-all. Project 44 changes the surface problem while preserving the underlying method.

progress check

Reading target: strip comments before parsing rules is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

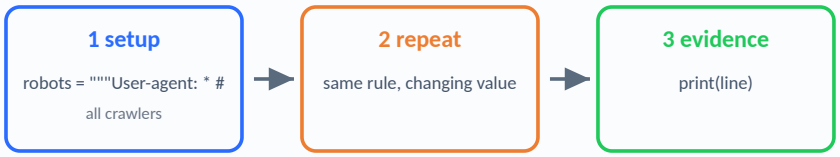
Logic Lens

- Setup: `robots = """User-agent: * # all crawlers` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(line)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: * # all crawlers`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(line)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Strip Comments Before Parsing Rules: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, all, crawlers. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * # all crawlers | Core: if line: | Evidence: print(line)

🛠️ Reader to Writer Bridge

1. Read: explain strip comments before parsing rules in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 42.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 44 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(line)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: * # all crawlers` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of strip comments before parsing rules and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Cutting each line at its first `#` and stripping whitespace removes both full-line and trailing comments, leaving only the directive text to parse.

```
robots = """User-agent: *  # all crawlers
Disallow: /admin  # back office
# whole-line comment
Allow: /public
"""
cleaned = []
for line in robots.splitlines():
    line = line.split('#', 1)[0].strip()
    if line:
        cleaned.append(line)
for line in cleaned:
    print(line)
```

```
output
User-agent: *
Disallow: /admin
Allow: /public
```

How it works:

1. The robots string carries inline comments after directives and one whole-line comment.
2. Each line is split on its first '#' and only the portion before it is kept.
3. That portion is stripped, and non-empty results are appended to cleaned.
4. The whole-line comment vanishes and the trailing notes are trimmed, printing three bare directives.

Try it:

1. Feed a line that is only whitespace before a comment and confirm it is dropped
2. Preserve a hash that appears inside a quoted value rather than treating it as a comment
3. Count how many comment lines were removed and print that total
4. Add a path that legitimately contains `#fragment` and see it get truncated.

⚠ Common Mistakes

- Splitting on `#` without `maxsplit=1` is harmless here but discards the rest if you later need it.
- A `#` inside a quoted value would be cut too; robots.txt has no quoting, so this is safe but worth knowing.
- Dropping the empty-after-strip check keeps blank lines that the parser must then skip anyway.

Skills gained comment stripping | str.split limit | line cleaning

→ Where You'll Use This: Comment removal is a mandatory preprocessing step in robots parsers, since real files routinely document their rules with hash comments a naive parser would mishandle.

Mental Model

The hash is an editor's red pen: everything to the right of it on the line is struck out before the document goes to the parser.

Project 44 Index Every User-Agent Group Separately

The Idea Behind It

A robots.txt can name many bots, each carrying its own block of rules. This project parses the file into a dictionary keyed by user-agent, skipping blanks and attaching rules to whichever agent is currently open. It then prints the sorted set of agents and the rule count under each, so you can answer questions per bot rather than treating the file as one undifferentiated list.

What You'll Build & Learn

Index every user-agent block into its own dictionary entry and report how many rules each bot has.

- one file can target many bots
- key rules by user-agent
- count rules per group

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 43: strip comments before parsing rules. Project 45 changes the surface problem while preserving the underlying method.

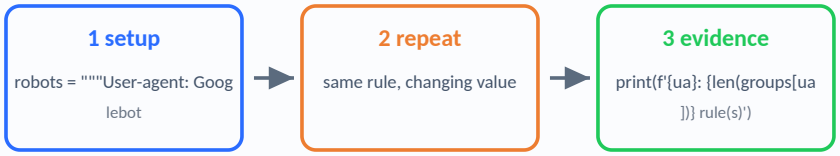
progress check

Reading target: index every user-agent group separately is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `robots = """User-agent: Googlebot` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{ua}: {len(groups[ua])} rule(s)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: Googlebot`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{ua}: {len(groups[ua])} rule(s)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Index Every User-Agent Group Separately: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Googlebot, Disallow. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: Googlebot | Core: if not line: | Evidence: print(f'{ua}: {len(groups[ua])} rule(s)')

🛠️ Reader to Writer Bridge

1. Read: explain index every user-agent group separately in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 43.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 45 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{ua}: {len(groups[ua])} rule(s)')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `robots = """User-agent: Googlebot` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of index every user-agent group separately and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each `User-agent` line starts a fresh keyed list and the directives below it accumulate into that list, so the dict ends up holding one rule group per bot.

```
robots = """User-agent: Googlebot
Disallow: /no-google

User-agent: Bingbot
Disallow: /no-bing
```

```
User-agent: *
Disallow: /private
"""
groups, agent = {}, None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    field, _, value = line.partition(':')
    field, value = field.strip().lower(), value.strip()
    if field == 'user-agent':
        agent = value
        groups.setdefault(agent, [])
    elif agent:
        groups[agent].append((field, value))
print('groups found:', sorted(groups))
for ua in sorted(groups):
    print(f'{ua}: {len(groups[ua])} rule(s)')
```

```
output
groups found: ['*', 'Bingbot', 'Googlebot']
*: 1 rule(s)
Bingbot: 1 rule(s)
Googlebot: 1 rule(s)
```

How it works:

1. The robots string defines Googlebot, Bingbot, and wildcard blocks separated by blank lines.
2. Lines are stripped, blanks skipped, and fields partitioned and lowercased.
3. A user-agent line opens a new dictionary key; other lines append to the current agent's list.
4. The code prints the sorted agent names and then each agent with its rule count of one.

Try it:

1. Add a second Disallow under Bingbot and confirm its rule count rises to two
2. Report the total rule count across all groups in addition to the per-bot counts
3. Detect and warn when a user-agent name appears with no rules beneath it
4. Add a second block for `Googlebot` and watch its rule count grow.

⚠ Common Mistakes

- Resetting the group dict on every blank line wipes earlier blocks.
- Using a plain dict without `setdefault` raises KeyError on the first rule of a new group.
- Sorting bots by insertion order instead of name makes the printed list nondeterministic.

Skills gained multi-bot indexing dict grouping rule counting

→ Where You'll Use This: Indexing per bot is the data model behind comparing how a site treats Googlebot versus aggressive scrapers, a routine part of crawl-management audits.

Mental Model

Think of a hotel front desk with one mail slot per guest: each bot has its own slot, and you check the slot that belongs to the bot you care about.

Project 45 Answer Can Googlebot Fetch This Path

The Idea Behind It

The practical question is binary: may this bot fetch this URL? This project parses groups, then defines can_fetch, which scans the active rules and keeps the action of the longest matching pattern, favouring Allow on ties via its combined length check. Wrapping group selection and longest-match precedence into one helper turns the whole file into a yes-or-no answer per path.

What You'll Build & Learn

Wrap parsing and precedence into a can_fetch helper that returns whether a bot may crawl a given path.

- crawlability reduces to yes or no
- longest match selects the action
- Allow wins on a tie

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 44: index every user-agent group separately. Project 46 changes the surface problem while preserving the underlying method.

progress check

Reading target: answer can googlebot fetch this path is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

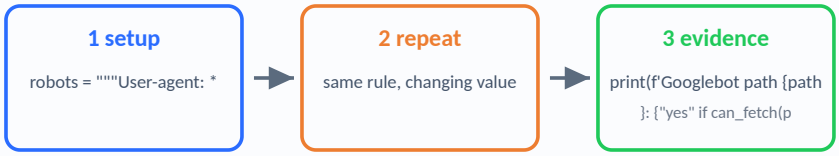
Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.

- Evidence: `print(f'Googlebot path {path}: {"yes" if can_fetch(path, rules) else "no"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'Googlebot path {path}: {"yes" if can_fetch(path, rules) else "no"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Answer Can Googlebot Fetch This Path: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, admin. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * | Core: if not line: | Evidence: print(f'Googlebot path {path}: {"yes" if can_fetch(path, rules) else "n...

🛠️ Reader to Writer Bridge

1. Read: explain answer can googlebot fetch this path in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 44.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 46 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'Googlebot path {path}: {"yes" if can_fetch(path, rules) else "no"}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of answer can googlebot fetch this path and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

`can_fetch` finds the longest matching rule (Allow winning equal-length ties) and returns whether that rule allows the path, giving a clean per-URL crawl decision.

```
robots = """User-agent: *
Disallow: /admin
Disallow: /cart
Allow: /cart/wishlist
"""
groups, agent = {}, None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    f, _, v = line.partition(':')
    f, v = f.strip().lower(), v.strip()
    if f == 'user-agent':
        agent = v; groups.setdefault(agent, [])
    elif agent:
        groups[agent].append((f, v))
def can_fetch(path, rules):
    best_len, action = -1, 'allow'
    for a, p in rules:
        if p and path.startswith(p) and len(p) >= best_len:
            if len(p) > best_len or a == 'allow':
                best_len, action = len(p), a
    return action == 'allow'
rules = groups.get('Googlebot', groups['*'])
for path in ['/blog', '/admin', '/cart', '/cart/wishlist']:
    print(f'Googlebot path {path}: {"yes" if can_fetch(path, rules) else
"no"}')
```

```
output
Googlebot path /blog: yes
Googlebot path /admin: no
Googlebot path /cart: no
Googlebot path /cart/wishlist: yes
```

How it works:

1. The file is parsed into groups with the now-familiar partition loop.
2. can_fetch scans rules, and a non-empty pattern the path starts with updates best_len when it is longer or an equal-length Allow.
3. rules is taken from the Googlebot group if present, else the wildcard.
4. /cart is blocked but /cart/wishlist crawls because its longer Allow beats the /cart Disallow.

Try it:

1. Pass a list of paths and print a crawlable count alongside each verdict

2. Add an early return when rules is empty so unrestricted bots short-circuit to yes
3. Extend the patterns with a wildcard and confirm can_fetch still resolves correctly
4. Add `Allow: /admin/help` and confirm that one sub-path becomes fetchable.

⚠ Common Mistakes

- Skipping the `if p` guard lets an empty rule string match every path with length 0.
- Using `groups['Googlebot']` directly raises KeyError when only a `*` group exists; `.get` with a fallback avoids that.
- Returning on the first matching rule instead of the longest gives wrong answers for nested paths.

Skills gained can_fetch helper | longest-match | Allow tie-break

→ Where You'll Use This: A can_fetch function is the public API of a robots library, called by crawlers and SEO tools to gate every URL fetch before a request goes out.

Mental Model

can_fetch is a turnstile that swallows the whole rulebook and lights up green or red for each URL you push through it.

Project 46 Count Allowed Versus Blocked URLs

The Idea Behind It

An audit over a URL list should summarise the damage: how many URLs stay crawlable and how many a Disallow set hides. This project counts allowed and blocked URLs with the same prefix test, then computes a block rate as a percentage of the total. That single number is a quick health signal for how aggressively a robots.txt is pruning the crawlable surface.

What You'll Build & Learn

Tally allowed and blocked URLs against a Disallow set and report the resulting block-rate percentage.

- audits summarise crawl coverage
- tally allowed versus blocked
- block rate as one signal

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 45: answer can googlebot fetch this path. Project 47 changes the surface problem while preserving the underlying method.

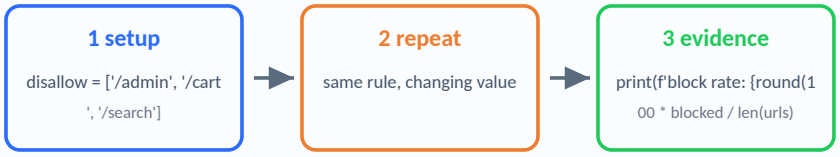
progress check

Reading target: count allowed versus blocked urls is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `disallow = ['/admin', '/cart', '/search']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'block rate: {round(100 * blocked / len(urls), 1)}%')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`disallow = ['/admin', '/cart', '/search']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if any(u.startswith(r) for r in disallow):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'block rate: {round(100 * blocked / len(urls), 1)}%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if any(u.startswith(r) for r in disallow):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Allowed Versus Blocked URLs: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: disallow, admin, cart, search, urls. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: disallow = ['/admin', '/cart', '/search'] | Core: if any(u.startswith(r) for r in disallow): | Evidence: print(f'block rate: {round(100 * blocked / len(urls), 1)}%')

🛠️ Reader to Writer Bridge

1. Read: explain count allowed versus blocked urls in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 45.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 47 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'block rate: {round(100 * blocked / len(urls), 1)}%')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `disallow = ['/admin', '/cart', '/search']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count allowed versus blocked urls and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Tallying each URL against the Disallow prefixes splits the list into allowed and blocked counts, and the ratio gives a block-rate percentage.

```
disallow = ['/admin', '/cart', '/search']
urls = ['/', '/blog', '/admin/users', '/products', '/cart', '/search?q=x',
'/about']
allowed = blocked = 0
for u in urls:
    if any(u.startswith(r) for r in disallow):
        blocked += 1
    else:
        allowed += 1
print(f'allowed: {allowed}')
print(f'blocked: {blocked}')
print(f'block rate: {round(100 * blocked / len(urls), 1)}%')
```

```
output
allowed: 4
blocked: 3
block rate: 42.9%
```

How it works:

1. disallow lists three blocked prefixes and urls lists seven candidate paths.
2. allowed and blocked counters start at zero.
3. For each URL, any() with startswith increments blocked on a match and allowed otherwise.
4. The code prints both counts and round(100 * blocked / total, 1), yielding a 42.9% block rate.

Try it:

1. Print the list of blocked URLs so the report names them, not just a count
2. Add an Allow override and account for it before tallying each URL
3. Guard against an empty URL list so the percentage does not divide by zero
4. Add `/search/help` to an Allow list and exclude it from the blocked tally.

⚠ Common Mistakes

- Counting URLs that merely contain a rule substring instead of starting with it inflates the blocked count.

- Dividing integers without `* 100` or floats gives 0 for the percentage in older code.
- Double-counting a URL matched by two rules; `any` short-circuits to count it once.

Skills gained block-rate metric counting loop percentage calc

→ **Where You'll Use This:** A block-rate summary is the headline number in a crawl-coverage report, flagging at a glance when a robots.txt is hiding far more of a site than intended.

Mental Model

It is a quick census of the site's doors: count how many are open versus locked, then quote the locked fraction as a one-line headline.

Project 47 Parse Meta Robots from an HTML Head

The Idea Behind It

Beyond robots.txt, a page can carry a meta robots tag whose content lists directives like noindex and nofollow. This project uses a regex to pull the content value out of the HTML head, splits it on commas into a lowercased set, and derives whether the page is indexable and whether its links are followed. It tells you if a crawlable page actually wants to be indexed.

What You'll Build & Learn

Extract meta robots directives from an HTML head and report indexability and link-following status.

- meta robots adds page-level control
- regex extracts the content value
- split directives into a set

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 46: count allowed versus blocked urls. Project 48 changes the surface problem while preserving the underlying method.

progress check

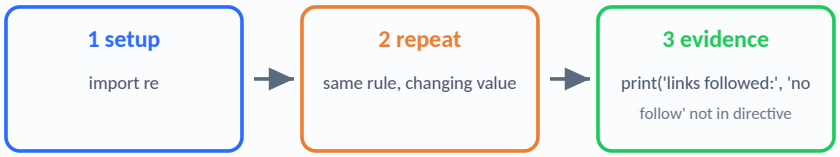
Reading target: parse meta robots from an html head is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.

- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print('links followed:', 'nofollow' not in directives)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if m:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('links followed:', 'nofollow' not in directives)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Parse Meta Robots from an HTML Head: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, html, head, meta, name. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if m: | Evidence: print('links followed:', 'nofollow' not in directives)

🛠️ Reader to Writer Bridge

1. Read: explain parse meta robots from an html head in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 46.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 48 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('links followed:', 'nofollow' not in directives)`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of parse meta robots from an html head and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A regex pulls the `content` attribute of the robots meta tag, then splitting on commas and lowercasing yields the directive set used to answer index/follow questions.

```
import re
html = '<head><meta name="robots" content="noindex, nofollow"><title>X</title>
</head>'
m = re.search(r'<meta\s+name=["\']robots["\']\s+content=["\']([^"\']+)["\']',
html, re.I)
directives = set()
if m:
    directives = {d.strip().lower() for d in m.group(1).split(',')}
print('directives:', sorted(directives))
print('indexable:', 'noindex' not in directives)
print('links followed:', 'nofollow' not in directives)
```

```
output
directives: ['nofollow', 'noindex']
indexable: False
links followed: False
```

How it works:

1. html holds a head with a meta robots tag whose content is noindex, nofollow.
2. re.search with a case-insensitive flag captures the content attribute value.
3. The captured string is split on commas into a lowercased directives set.
4. indexable is 'noindex' not in directives and links followed is 'nofollow' not in directives, both False here.

Try it:

1. Handle a content value of index, follow and confirm both derived flags become True
2. Capture a googlebot-scoped meta tag separately from the generic robots tag
3. Detect a noarchive or nosnippet directive and print its status too
4. Add `noarchive` to the content and report it separately.

⚠ Common Mistakes

- A greedy or unanchored regex can grab the wrong meta tag when several exist.
- Not lowercasing leaves `NoIndex` unmatched against the literal `noindex` check.

- Confusing meta robots (page must be crawled to read it) with robots.txt (blocks the crawl entirely).

Skills gained meta robots parsing | regex capture | set membership

→ Where You'll Use This: Meta robots extraction is core to page-level SEO audits, catching pages set to noindex that still appear in sitemaps or internal links by mistake.

Mental Model

If robots.txt is the doorman deciding who enters, the meta robots tag is a note on the page itself saying whether it wants its photo in the directory.

Project 48 Read Index Rules from an X-Robots-Tag Header

The Idea Behind It

Non-HTML files such as PDFs cannot carry a meta tag, so their indexing rules ride in the X-Robots-Tag HTTP header instead. This project reads that header from a headers dict, splits its value into a lowercased set, and checks a list of common directives against it. Parsing the header gives the same noindex and noarchive controls a meta tag would for HTML pages.

What You'll Build & Learn

Parse the X-Robots-Tag response header into a directive set and report which index controls are active.

- headers control non-HTML indexing
- X-Robots-Tag mirrors meta robots
- split a header into a set

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 47: parse meta robots from an html head. Project 49 changes the surface problem while preserving the underlying method.

progress check

Reading target: read index rules from an x-robots-tag header is the surface task; the transferable Python move is to mark setup, rule, and output before running.

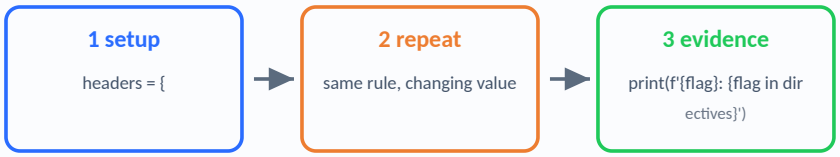
setup -> rule -> result

Logic Lens

- Setup: `headers = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.

- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{flag}: {flag in directives}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`headers = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`directives = {d.strip().lower() for d in tag.split(',') if d.strip()}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{flag}: {flag in directives}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `directives = {d.strip().lower() for d in tag.split(',') if d.strip()}`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Read Index Rules from an X-Robots-Tag Header: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: headers, Content, Type, application, pdf. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: headers = { | Core: directives = {d.strip().lower() for d in tag.split(',') if d.strip()} | Evidence: print(f'{flag}: {flag in directives}')

🛠️ Reader to Writer Bridge

1. Read: explain read index rules from an x-robots-tag header in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 47.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 49 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{flag}: {flag in directives}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `headers = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of read index rules from an x-robots-tag header and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The header value is a comma-separated directive list; splitting and lowercasing it into a set lets a simple membership test answer each indexing question.

```
headers = {
    'Content-Type': 'application/pdf',
    'X-Robots-Tag': 'noindex, noarchive',
}
tag = headers.get('X-Robots-Tag', '')
directives = {d.strip().lower() for d in tag.split(',') if d.strip()}
print('X-Robots-Tag directives:', sorted(directives))
for flag in ['noindex', 'nofollow', 'noarchive', 'nosnippet']:
    print(f'{flag}: {flag in directives}')
```

```
output
X-Robots-Tag directives: ['noarchive', 'noindex']
noindex: True
nofollow: False
noarchive: True
nosnippet: False
```

How it works:

1. headers includes a PDF content type and an X-Robots-Tag of noindex, noarchive.
2. tag reads the header value, defaulting to an empty string when absent.
3. The value is split on commas, stripped, lowercased, and emptied entries dropped, forming directives.
4. Looping over four candidate flags prints noindex and noarchive as True, nofollow and nosnippet as False.

Try it:

1. Add nosnippet to the header value and confirm its flag turns True
2. Handle a missing X-Robots-Tag header and print a sensible default state
3. Combine multiple header values, as some servers send the tag more than once
4. Add `nosnippet` to the header value and watch its flag flip True.

⚠ Common Mistakes

- Looking up the header with a case-sensitive key when real headers vary in case.
- Forgetting `if d.strip()` lets a trailing comma add an empty directive to the set.

- Assuming the header only carries one directive; it is a comma-separated list like the meta tag.

Skills gained X-Robots-Tag parsing | header dict | directive set

→ Where You'll Use This: X-Robots-Tag parsing matters whenever you audit PDFs, images, or API responses, where header directives are the only way to keep non-HTML assets out of the index.

Mental Model

A PDF has no margins to scribble in, so the noindex instruction is tucked into the delivery envelope's header rather than written on the page.

Project 49 Flag Rules That Block CSS and JS

The Idea Behind It

Blocking /css/ or /js/ stops Google from rendering a page the way users see it, which can hurt rankings. This project collects every Disallow value, then flags any whose path mentions css or js or ends in those extensions. Auditing the Disallow list for render-critical resources is a common and high-impact SEO fix that a quick scan can surface automatically.

What You'll Build & Learn

Scan the Disallow list for blocked CSS or JS paths that would stop Google rendering the page.

- blocking CSS or JS breaks rendering
- render gaps hurt rankings
- scan disallows for assets

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 48: read index rules from an x-robots-tag header. Project 50 changes the surface problem while preserving the underlying method.

progress check

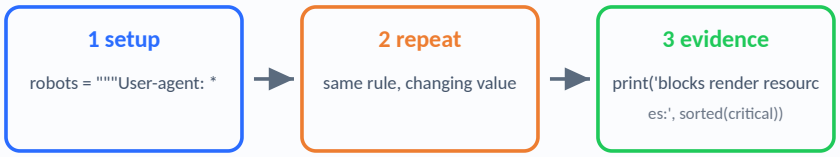
Reading target: flag rules that block css and js is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print('blocks render resources:', sorted(critical))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if f.strip().lower() == 'disallow':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('blocks render resources:', sorted(critical))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if f.strip().lower() == 'disallow':`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Rules That Block CSS and JS: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, css. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: * | Core: if f.strip().lower() == 'disallow': | Evidence: print('blocks render resources:', sorted(critical))

🛠️ Reader to Writer Bridge

1. Read: explain flag rules that block css and js in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 48.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 50 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('blocks render resources:', sorted(critical))`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag rules that block css and js and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Scanning each Disallow value for `css`/`js` markers (or `.css`/`.js` suffixes) flags the rules that would starve the renderer of stylesheets and scripts.

```
robots = """User-agent: *
Disallow: /css/
Disallow: /js/
Disallow: /admin
"""
blocked = []
for line in robots.splitlines():
    f, _, v = line.partition(':')
    if f.strip().lower() == 'disallow':
        blocked.append(v.strip())
critical = []
for path in blocked:
    low = path.lower()
    if 'css' in low or 'js' in low or low.endswith('.css') or
low.endswith('.js'):
        critical.append(path)
print('blocked paths:', blocked)
print('blocks render resources:', sorted(critical))
```

```
output
blocked paths: ['/css/', '/js/', '/admin']
blocks render resources: ['/css/', '/js/']
```

How it works:

1. The robots string disallows /css/, /js/, and /admin under the wildcard agent.
2. Each Disallow value is collected into the blocked list.
3. For each path, a lowercased copy is tested for css or js substrings or matching extensions.
4. Matching paths go into critical, so the code reports /css/ and /js/ as render-blocking, leaving /admin out.

Try it:

1. Extend the check to fonts and images so /fonts/ and /img/ are also flagged
2. Avoid a false positive on a path like /jsmith by requiring a folder or extension boundary
3. Print a remediation hint suggesting an Allow override for each flagged resource
4. Add `/assets/app.js` and confirm the suffix branch catches it.

⚠ Common Mistakes

- Matching only file extensions misses directory blocks like `/css/` that contain no dot.
- A path like `/justice` contains 'js' and would false-positive; tighten the test for production.
- Forgetting to lowercase means `/CSS/` slips past the marker check.

Skills gained render-resource audit substring test extension check

→ **Where You'll Use This:** Flagging blocked CSS and JS is a staple of technical-SEO audits, since render-blocking robots rules are a recurring cause of mobile-usability and ranking issues.

⚡ Mental Model

Blocking the CSS and JS folders is like handing Google a page with the stylesheet torn out: it sees raw bones, not the page real visitors load.

Project 50 Flag Unknown robots.txt Directives

💡 The Idea Behind It

Crawlers silently ignore directives they do not recognise, so a typo like Craw-delay is simply dropped with no error. This project defines a set of known fields and prints every line whose field is not in it. Listing the unknown directives surfaces these dead lines, including misspellings and unsupported keywords, before they cause confusion about why a rule never takes effect.

What You'll Build & Learn

List robots.txt fields that are not recognised so typos and unsupported directives become visible.

- unknown directives are silently ignored
- a typo becomes a dead line
- validate fields against a known set

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 49: flag rules that block css and js. Project 51 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag unknown robots.txt directives is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.

- Evidence: `print(f'unknown directive: {field}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'unknown directive: {field}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Unknown robots.txt Directives: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, admin. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: robots = """User-agent: * | Core: if not line: | Evidence: print(f'unknown directive: {field}')

🛠️ Reader to Writer Bridge

1. Read: explain flag unknown robots.txt directives in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 49.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 51 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'unknown directive: {field}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag unknown robots.txt directives and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Comparing each line's field name against a set of standard directives reveals typos and non-standard extensions that Googlebot will skip.

```
robots = """User-agent: *
Disallow: /admin
Craw-delay: 5
Host: example.com
Allow: /public
Visit-time: 0600-0845
"""
known = {'user-agent', 'disallow', 'allow', 'sitemap', 'crawl-delay'}
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    field = line.partition(':')[0].strip().lower()
    if field not in known:
        print(f'unknown directive: {field}')
```

```
output
unknown directive: craw-delay
unknown directive: host
unknown directive: visit-time
```

How it works:

1. known holds the recognised field names like user-agent, disallow, and crawl-delay.
2. Each line is stripped, blanks skipped, and its field extracted, stripped, and lowercased.
3. When the field is not in the known set, the code prints it as an unknown directive.
4. The misspelled craw-delay plus host and visit-time are reported as unknown.

Try it:

1. Add the line number to each warning so the typo is easy to locate
2. Suggest the closest known field for an unknown one using a string-distance check
3. Treat host as known-but-deprecated and label it differently from a true typo
4. Add `Clean-param` (a Yandex directive) and decide whether to allow it.

⚠ Common Mistakes

- Treating non-standard but real directives (like `Host` for Yandex) as errors rather than warnings.
- Not lowercasing the field lets `Sitemap` register as unknown against a lowercase known-set.
- Flagging blank lines as unknown directives; skip empty lines first.

Skills gained directive validation | set membership | typo detection

→ Where You'll Use This: Flagging unknown directives is a core robots-linter feature, catching the silent typos and legacy fields that leave an intended rule doing nothing in production.

⚡ Mental Model

The parser is a customs officer with a fixed list of valid forms: hand it a misspelled one and it does not argue, it just sets it aside unread.

Project 51 Respect Case-Sensitive robots.txt Paths

💡 The Idea Behind It

Directive names in robots.txt are case-insensitive, but the path values are case-sensitive, so Disallow: /Admin does not block /admin. This project tests several differently-cased paths against a single capitalised prefix using startswith. Confusing the two causes either over-blocking or under-blocking, and seeing /admin crawl while /Admin blocks makes the distinction concrete.

What You'll Build & Learn

Show that Disallow paths are case-sensitive so only exactly-cased paths match the prefix.

- directive names are case-insensitive
- path values are case-sensitive
- case mismatches misblock URLs

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 50: flag unknown robots.txt directives. Project 52 changes the surface problem while preserving the underlying method.

progress check

Reading target: respect case-sensitive robots.txt paths is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

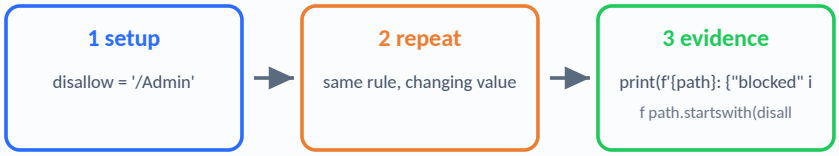
- Setup: `disallow = '/Admin'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`disallow = '/Admin'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Respect Case-Sensitive robots.txt Paths: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: disallow, Admin, paths, admin, login. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: disallow = '/Admin' | Core: print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}') | Evidence: print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}')

🛠️ Reader to Writer Bridge

1. Read: explain respect case-sensitive robots.txt paths in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 50.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 52 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `disallow = '/Admin'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of respect case-sensitive robots.txt paths and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Plain `str.startswith` is case-sensitive, so the `/Admin` rule blocks `/Admin` and `/Admin/users` but leaves the lowercase `/admin` and all-caps `/ADMIN/login` crawlable.

```
disallow = '/Admin'
paths = ['/Admin', '/admin', '/ADMIN/login', '/Admin/users']
```

```
    for path in paths:
        print(f'{path}: {"blocked" if path.startswith(disallow) else "crawl"}')
```

```
output
/Admin: blocked
/admin: crawl
/ADMIN/login: crawl
/Admin/users: blocked
```

How it works:

1. disallow holds the capitalised prefix /Admin.
2. paths lists four variants differing only in letter case.
3. Each path is tested with path.startswith(disallow) without any case folding.
4. /Admin and /Admin/users match and block, while lowercase /admin and uppercase /ADMIN/login crawl.

Try it:

1. Add a second rule for /admin and confirm both cases are now blocked
2. Implement a case-insensitive variant and discuss why it diverges from the spec
3. Test a mixed-case path like /Admin/Users to confirm only the prefix case matters
4. Lowercase both sides and watch every variant get blocked.

⚠ Common Mistakes

- Lowercasing the path before comparison wrongly blocks `/admin` and `/ADMIN/login`.
- Assuming the whole robots.txt is case-insensitive; only directive keywords are, not the paths.
- Expecting the rule to behave like a case-insensitive filter; `/admin` slips through while `/Admin/users` is correctly caught.

Skills gained case sensitivity | startswith | path matching

→ Where You'll Use This: Case sensitivity catches subtle robots bugs where a rule written in the wrong case fails to protect the intended URLs or accidentally blocks a different casing.

Mental Model

URL paths are like Unix filenames: Admin and admin are two different doors, so a rule guarding one leaves the other wide open.

Project 52 Handle Trailing Slashes in Disallow Rules

The Idea Behind It

Disallow: /blog and Disallow: /blog/ are not the same rule. This project tests four paths against both a slashless and a slashed prefix, printing each result. The slashless form also catches /blogger because the prefix matches, while the slashed form is confined to the directory and its children. The trailing slash narrows the match to a folder boundary.

What You'll Build & Learn

Contrast a slashless and a slashed Disallow prefix to show how a trailing slash narrows the match.

- a trailing slash changes the match
- slashless catches sibling prefixes
- slashed confines to a directory

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 51: respect case-sensitive robots.txt paths. Project 53 changes the surface problem while preserving the underlying method.

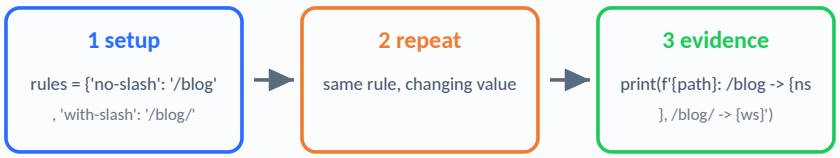
progress check

Reading target: handle trailing slashes in disallow rules is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `rules = {'no-slash': '/blog', 'with-slash': '/blog/'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: /blog -> {ns}, /blog/ -> {ws}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`rules = {'no-slash': '/blog', 'with-slash': '/blog/'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`for path in paths:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: /blog -> {ns}, /blog/ -> {ws}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for path in paths:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Handle Trailing Slashes in Disallow Rules: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: rules, no, slash, blog, paths. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: rules = {'no-slash': '/blog', 'with-slash': '/blog/'} | Core: for path in paths: | Evidence: print(f'{path}: /blog -> {ns}, /blog/ -> {ws}')

🛠️ Reader to Writer Bridge

1. Read: explain handle trailing slashes in disallow rules in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 51.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 53 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: /blog -> {ns}, /blog/ -> {ws}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `rules = {'no-slash': '/blog', 'with-slash': '/blog/'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of handle trailing slashes in disallow rules and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Prefix matching means `/blog` matches anything starting with those five characters, including `/blogger`, whereas `/blog/` requires the slash and so excludes the unrelated `/blogger` path.

```
rules = {'no-slash': '/blog', 'with-slash': '/blog/'}
paths = ['/blog', '/blog/', '/blog/post', '/blogger']
for path in paths:
    ns = path.startswith(rules['no-slash'])
    ws = path.startswith(rules['with-slash'])
    print(f'{path}: /blog -> {ns}, /blog/ -> {ws}')
```

```
output
/blog: /blog -> True, /blog/ -> False
/blog/: /blog -> True, /blog/ -> True
```

```
/blog/post: /blog -> True, /blog/ -> True
/blogger: /blog -> True, /blog/ -> False
```

How it works:

1. rules maps 'no-slash' to /blog and 'with-slash' to /blog/.
2. paths includes /blog, /blog/, /blog/post, and the lookalike /blogger.
3. Each path is tested against both prefixes with startswith into ns and ws.
4. /blogger matches /blog but not /blog/, exposing how the slash blocks the false sibling match.

Try it:

1. Add /blog-archive to the paths and observe which prefix catches it
2. Decide and implement which form correctly blocks only the blog directory
3. Report a warning when a slashless rule would catch an unintended sibling
4. Add `/blog-archive` and see which rule catches it.

⚠ Common Mistakes

- Assuming `/blog` only blocks the blog section; it also blocks lookalikes like `/blogger`.
- Thinking `/blog/` blocks the bare `/blog` URL; it does not, since `/blog` lacks the trailing slash.
- Adding a trailing slash to normalize paths changes which rules match.

Skills gained trailing-slash semantics | prefix comparison | boundary matching

→ Where You'll Use This: Understanding the trailing slash prevents a common audit error where a folder rule unexpectedly blocks sibling pages whose names share the same opening characters.

Mental Model

Without the slash, /blog is a prefix net that scoops up /blogger too; adding the slash draws the boundary right at the folder wall.

Project 53 Default to Allow When No Rule Matches

The Idea Behind It

robots.txt is permissive by default: a path the file never mentions stays crawlable. This project tests each path against both a Disallow set and an Allow set, marking a URL blocked only when a Disallow matches and no Allow does. Modelling the default-allow rule explicitly stops an audit from wrongly flagging unlisted URLs, which crawlers are free to fetch, as blocked.

What You'll Build & Learn

Apply default-allow so a path is blocked only when a Disallow matches and no Allow overrides it.

- unmentioned paths are crawlable
- default behaviour is allow
- Allow can override a Disallow

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 52: handle trailing slashes in disallow rules. Project 54 changes the surface problem while preserving the underlying method.

progress check

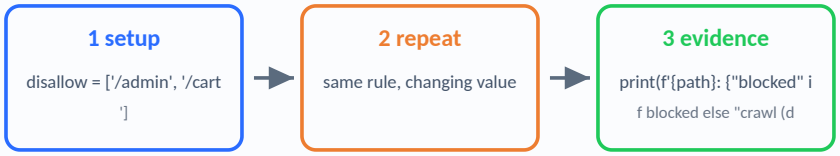
Reading target: default to allow when no rule matches is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `disallow = ['/admin', '/cart']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`disallow = ['/admin', '/cart']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.

Move	Code to inspect	Why it matters
2. Apply the rule	`print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Default to Allow When No Rule Matches: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: disallow, admin, cart, allow, paths. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: disallow = ['/admin', '/cart'] | Core: print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}') | Evidence: print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}')

Reader to Writer Bridge

1. Read: explain default to allow when no rule matches in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 52.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 54 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `disallow = ['/admin', '/cart']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of default to allow when no rule matches and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A path is blocked only when a Disallow matches and no Allow rescues it; everything else, including unlisted paths, falls through to the default-allow.

```
disallow = ['/admin', '/cart']
allow = ['/cart/public']
paths = ['/blog', '/contact', '/admin', '/cart/public']
for path in paths:
    d = any(path.startswith(r) for r in disallow)
    a = any(path.startswith(r) for r in allow)
    blocked = d and not a
    print(f'{path}: {"blocked" if blocked else "crawl (default allow)"}')
```

```
output
/blog: crawl (default allow)
/contact: crawl (default allow)
/admin: blocked
/cart/public: crawl (default allow)
```

How it works:

1. disallow lists two blocked prefixes and allow lists one override sub-path.
2. For each path, d is whether any Disallow matches and a whether any Allow matches.

3. blocked is d and not a, so a matched Allow rescues the path.
4. /blog and /contact are never listed and crawl under default allow; /admin blocks; /cart/public crawls via its Allow.

Try it:

1. Add a /search URL and confirm it crawls because the file never mentions it
2. Switch to longest-match resolution and compare results against this simpler model
3. Label each verdict with whether it came from a rule or the default
4. Remove the Allow list and watch `/cart/public` flip to blocked.

⚠ Common Mistakes

- Defaulting unmatched paths to blocked inverts robots.txt's permissive model.
- Ignoring the Allow list lets `/cart/public` get blocked by the `/cart` Disallow.
- Using `d or not a` instead of `d and not a` blocks far too much.

Skills gained default-allow model | Allow override | boolean logic

→ Where You'll Use This: Encoding the default-allow rule keeps crawl audits honest, since most of a site's URLs are unlisted and must be reported as crawlable rather than uncertain.

Mental Model

Picture a park that is open unless a sign forbids entry: no sign means walk right in, so any path the file forgot to mention is fair game.

Project 54 Check robots.txt Against the Size Limit

The Idea Behind It

Google parses only the first 500 KiB of a robots.txt and ignores anything past that, so a bloated file can have later rules silently dropped. This project measures the byte size with utf-8 encoding, compares it against the limit, and reports whether the file fits, plus its line count. Measuring the size catches an oversized file whose tail-end directives stop applying.

What You'll Build & Learn

Measure a robots.txt byte size against Google's 500 KiB limit and report whether it fits.

- Google parses only the first 500 KiB
- rules past the limit are ignored
- measure size in bytes not characters

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 53: default to allow when no rule matches. Project 55 changes the surface problem while preserving the underlying method.

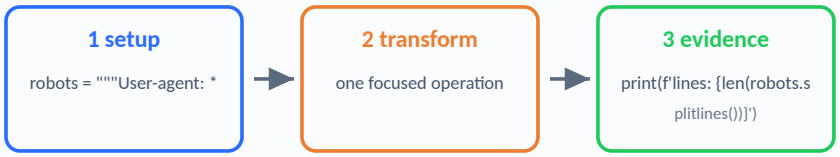
progress check

Reading target: check robots.txt against the size limit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `robots = """User-agent: *` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is deliberately small: Python stores a value, applies one clear operation, and prints evidence. That tiny chain is the reading pattern larger programs still use.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'lines: {len(robots.splitlines())}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: *`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`Disallow: /admin`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(f'lines: {len(robots.splitlines())}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `Disallow: /admin`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Check robots.txt Against the Size Limit: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Disallow, admin. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: robots = """User-agent: * | Core: Disallow: /admin | Evidence: print(f'lines: {len(robots.splitlines())}')

Reader to Writer Bridge

1. Read: explain check robots.txt against the size limit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 53.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 55 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'lines: {len(robots.splitlines())}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `robots = """User-agent: *` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check robots.txt against the size limit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Encoding the text to UTF-8 and taking its length gives the true byte size (not character count), which is what the 500 KiB parse limit applies to.

```
robots = """User-agent: *
Disallow: /admin
Disallow: /private
Sitemap: https://example.com/sitemap.xml
"""
size = len(robots.encode('utf-8'))
limit = 500000
print(f'robots.txt size: {size} bytes')
print(f'within {limit}-byte limit: {size <= limit}')
print(f'lines: {len(robots.splitlines())}')
```

```
output
robots.txt size: 91 bytes
within 500000-byte limit: True
lines: 4
```

How it works:

1. A small robots string holds two Disallow lines and a Sitemap.
2. size is len(robots.encode('utf-8')), the true byte length rather than the character count.
3. limit is 500000, and the code prints whether size is within it.
4. It also prints the line count, here 91 bytes across 4 lines, well within the cap.

Try it:

1. Build a large robots string and confirm the within-limit flag flips to False
2. Report the percentage of the limit consumed for an early-warning threshold
3. Account for multi-byte characters by adding non-ASCII text and rechecking the byte count
4. Multiply the robots text by 6000 and watch the limit check fail.

⚠ Common Mistakes

- Using `len(robots)` counts characters, not bytes; multibyte chars make the two differ.
- Confusing 500 KB (500000) with 500 KiB (512000); know which limit you are checking.
- Assuming rules past the limit error out; they are simply ignored, which is sneakier.

Skills gained byte-size check | utf-8 encoding | size limit

→ Where You'll Use This: A size check guards against generated robots files that grow past Google's limit, where the truncated tail quietly stops protecting or permitting URLs as intended.

⚡ Mental Model

Google reads robots.txt like a reader with a strict page budget: it stops at the 500 KiB mark, so anything written beyond that page is never seen.

Project 55 Merge Repeated Groups for the Same Bot

💡 The Idea Behind It

When a robots.txt names the same bot in two separate blocks, crawlers merge those rules into one group rather than letting the second block replace the first. This project parses the file with setdefault keyed by agent, so a repeated Googlebot key accumulates rules from both blocks. Merging is what makes split configurations behave, preserving every directive across the duplicated headers.

What You'll Build & Learn

Merge rules from two blocks naming the same bot into a single combined rule group.

- repeated bot blocks are merged
- the second block does not replace
- setdefault accumulates rules

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 54: check robots.txt against the size limit. Project 56 changes the surface problem while preserving the underlying method.

progress check

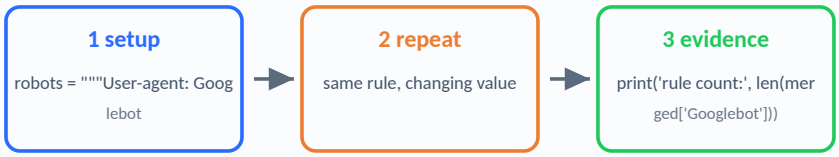
Reading target: merge repeated groups for the same bot is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `robots = """User-agent: Googlebot` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print('rule count:', len(merged['Googlebot']))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: Googlebot`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('rule count:', len(merged['Googlebot']))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Merge Repeated Groups for the Same Bot: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Googlebot, Disallow. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: Googlebot | Core: if not line: | Evidence: print('rule count:', len(merged['Googlebot']))

🛠️ Reader to Writer Bridge

1. Read: explain merge repeated groups for the same bot in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 54.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 56 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('rule count:', len(merged['Googlebot']))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `robots = """User-agent: Googlebot` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of merge repeated groups for the same bot and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Using `setdefault` keyed by bot name means a repeated `User-agent: Googlebot` reopens the existing list and appends to it, merging both blocks' rules.

```
robots = """User-agent: Googlebot
Disallow: /a

User-agent: Googlebot
Disallow: /b
Allow: /a/ok
"""
merged, agent = {}, None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    f, _, v = line.partition(':')
    f, v = f.strip().lower(), v.strip()
    if f == 'user-agent':
        agent = v
        merged.setdefault(agent, [])
    elif agent:
        merged[agent].append((f, v))
print('Googlebot merged rules:', merged['Googlebot'])
print('rule count:', len(merged['Googlebot']))
```

```
output
Googlebot merged rules: [('disallow', '/a'), ('disallow', '/b'), ('allow', '/a/ok')]
rule count: 3
```

How it works:

1. The robots string has two Googlebot blocks separated by a blank line.
2. On each user-agent line, setdefault only creates the list if the key is new, never resetting it.
3. Rules from both blocks append to the same Googlebot list.
4. The merged group prints all three rules with a final count of three.

Try it:

1. Add a third Googlebot block and confirm all its rules join the merged group

2. Detect that the bot appears more than once and warn about a possibly unintended split
3. Compare merged behaviour against a replace model and print the difference
4. Add a third Googlebot block and confirm the rule count climbs.

⚠ Common Mistakes

- Reassigning `groups[agent] = []` on each block overwrites the earlier rules instead of merging.
- Assuming the last block wins; the spec merges all blocks for the same agent.
- Forgetting that case-insensitive agent names mean `googlebot` and `Googlebot` should merge too.

Skills gained group merging | setdefault | duplicate handling

→ Where You'll Use This: Merging duplicate groups reflects real crawler behaviour, so an audit of a config split across edits reports the bot's true effective ruleset rather than just the last block.

Mental Model

Two blocks for the same bot are like two pages of one shopping list: the crawler staples them together instead of throwing the first page away.

Project 56 Decide Wildcard Precedence by Effective Length

The Idea Behind It

When wildcard rules compete, Google compares pattern lengths with the asterisks and dollar signs stripped out. This project compiles each pattern to a regex, and for the rules that match a path computes an effective length by removing the wildcard characters, keeping the longer one and favouring Allow on a tie. That effective length decides which of two matching wildcard rules wins.

What You'll Build & Learn

Pick the winning wildcard rule by comparing pattern lengths with wildcard characters removed.

- wildcard winners use effective length
- strip stars and dollars before measuring
- Allow breaks an effective-length tie

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 55: merge repeated groups for the same bot. Project 57 changes the surface problem while preserving the underlying method.

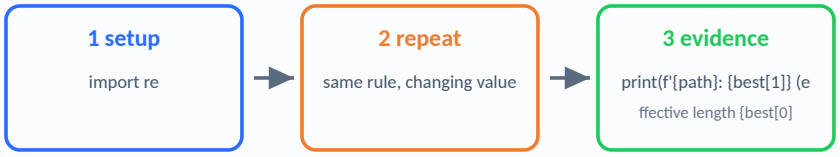
progress check

Reading target: decide wildcard precedence by effective length is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {best[1]} (effective length {best[0]})')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if ch == '*':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {best[1]} (effective length {best[0]})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if ch == '*':`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Decide Wildcard Precedence by Effective Length: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, compile_rule, pattern, rx, ch. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if ch == '*': | Evidence: print(f'{path}: {best[1]} (effective length {best[0]})')

🛠️ Reader to Writer Bridge

1. Read: explain decide wildcard precedence by effective length in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 55.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 57 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {best[1]} (effective length {best[0]})')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of decide wildcard precedence by effective length and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Stripping `*` and `$` before measuring length gives each pattern a comparable specificity, so `/folder/*.html` (effective 13) beats the shorter `/folder/page` (12) Allow.

```
import re
def compile_rule(pattern):
    rx = ['^']
    for ch in pattern:
        if ch == '*':
            rx.append('.*')
        elif ch == '$':
            rx.append('$')
        else:
            rx.append(re.escape(ch))
    return re.compile(''.join(rx))
rules = [('allow', '/folder/page'), ('disallow', '/folder/*.html')]
path = '/folder/page.html'
best = None
for action, pat in rules:
    if compile_rule(pat).match(path):
        eff = len(pat.replace('*', '').replace('$', ''))
        if best is None or eff > best[0] or (eff == best[0] and action ==
'allow'):
            best = (eff, action)
print(f'{path}: {best[1]} (effective length {best[0]})')
```

```
output
/folder/page.html: disallow (effective length 13)
```

How it works:

1. compile_rule turns a pattern into an anchored regex, mapping star to .* and dollar to $.
2. rules pairs an Allow /folder/page with a Disallow /folder/*.html.
3. For the path /folder/page.html, each matching pattern's effective length removes * and $ characters.
4. The Disallow's effective length of 13 beats the Allow, so the path is disallowed.

Try it:

1. Add a longer Allow pattern and confirm it flips the verdict to allow
2. Print each rule's effective length to make the comparison transparent

3. Test a path that matches neither rule and ensure best stays None safely
4. Lengthen the Allow to `/folder/page.htmlx` and watch it win.

⚠ Common Mistakes

- Measuring the raw pattern length counts the `*`, distorting the comparison.
- Forgetting the Allow-wins-ties branch flips genuine ties the wrong way.
- Using the matched path length instead of the pattern length compares the wrong thing.

Skills gained effective length | wildcard precedence | regex compile

→ Where You'll Use This: Effective-length comparison is how Google resolves dueling wildcard rules, so a faithful robots tester must use it to predict verdicts on patterned directives.

Mental Model

Stripping the asterisks before measuring is like ignoring elastic in a tape measure: you compare the fixed, literal length each pattern actually pins down.

Project 57 Build a Full Per-Bot Allow Evaluator

The Idea Behind It

This project combines everything into a miniature robots engine. It compiles each pattern to regex, picks the bot's group or falls back to the wildcard, and applies effective-length precedence with an Allow tie-break to return a single allow or block per path. The allowed function answers the real question for any bot and URL, exercising group fallback, wildcards, and longest-match together.

What You'll Build & Learn

Build a per-bot evaluator that selects a group, compiles patterns, and returns one allow-or-block verdict.

- a full engine combines all rules
- fall back to the wildcard group
- effective length plus Allow tie-break

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 56: decide wildcard precedence by effective length. Project 58 changes the surface problem while preserving the underlying method. *progress check*

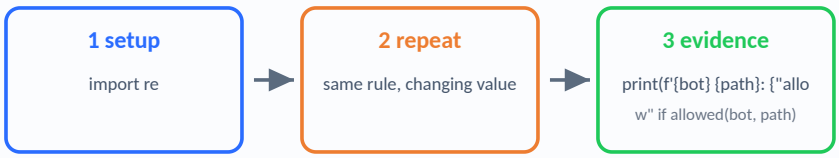
Reading target: build a full per-bot allow evaluator is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{bot} {path}: {"allow" if allowed(bot, path) else "block"}')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{bot} {path}: {"allow" if allowed(bot, path) else "block"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Build a Full Per-Bot Allow Evaluator: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, to_rx, pat, rx, ch. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: rx.append('.' *if ch ==* '' else ('$' if ch == '$' else re.escape(ch))) | Evidence: print(f'{bot} {path}: {"allow" if allowed(bot, path) else "block"}')

Reader to Writer Bridge

1. Read: explain build a full per-bot allow evaluator in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 56.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 58 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{bot} {path}: {"allow" if allowed(bot, path) else "block"}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a full per-bot allow evaluator and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Group selection plus regex matching plus effective-length precedence yields a per-bot decision, so Googlebot is freed for `/private/feed` while AhrefsBot inherits the `*` block-everything rule.

```
import re
def to_rx(pat):
    rx = ['^']
    for ch in pat:
        rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))
    return re.compile(''.join(rx))
groups = {
    'Googlebot': [('disallow', '/private'), ('allow', '/private/feed')],
    '*': [('disallow', '/')],
}
def allowed(bot, path):
    rules = groups.get(bot, groups.get('*', []))
    best_len, action = -1, 'allow'
    for a, pat in rules:
        if to_rx(pat).match(path):
            eff = len(pat.replace('*', '').replace('$', ''))
            if eff > best_len or (eff == best_len and a == 'allow'):
                best_len, action = eff, a
    return action == 'allow'
for bot, path in [('Googlebot', '/private'), ('Googlebot', '/private/feed'),
('AhrefsBot', '/home')]:
    print(f'{bot} {path}: {"allow" if allowed(bot, path) else "block"}')
```

```
output
Googlebot /private: block
Googlebot /private/feed: allow
AhrefsBot /home: block
```

How it works:

1. to_rx compiles a pattern to an anchored regex, mapping star and dollar.
2. groups defines a Googlebot block and a wildcard that disallows everything.

3. allowed picks the bot's rules or the wildcard, then keeps the longest effective-length match, favouring Allow on ties.
4. Googlebot reaches /private/feed via its Allow, but AhrefsBot falls to the wildcard Disallow: / and is blocked.

Try it:

1. Add a Bingbot group and confirm it is chosen ahead of the wildcard fallback
2. Return the winning rule with the verdict for easier debugging output
3. Cache compiled regexes so repeated paths do not recompile every pattern
4. Add a named `AhrefsBot` group and watch its verdict change.

⚠ Common Mistakes

- Forgetting the `*` fallback leaves unnamed bots with no rules and a wrong allow.
- Comparing raw pattern length instead of effective length mis-ranks wildcard rules.
- Recompiling the regex inside the loop is fine here but wasteful on large rule sets.

Skills gained robots engine | group fallback | longest-match

→ Where You'll Use This: A combined evaluator like this is the heart of a production robots library that crawlers and SEO platforms call to gate every fetch across many user-agents.

Mental Model

This is the whole assembly line in one machine: feed in a bot and a URL, and group selection, wildcard matching, and precedence stamp out a single green or red light.

Project 58 Trace Overlapping Rules to the Winner

The Idea Behind It

When several rules match one path, it helps to rank them by length before naming the winner. This project gathers every matching rule into tuples of length, action, and pattern, sorts them so the longest and Allow-favoured rule lands last, then reports the ranking and the winner. Making the order visible turns longest-match precedence from a black box into an explainable trace.

What You'll Build & Learn

List all rules matching a path ranked by length, then name the winner under longest-match precedence.

- rank matching rules by length
- sort surfaces the winner last
- make precedence explainable

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 57: build a full per-bot allow evaluator. Project 59 changes the surface problem while preserving the underlying method.

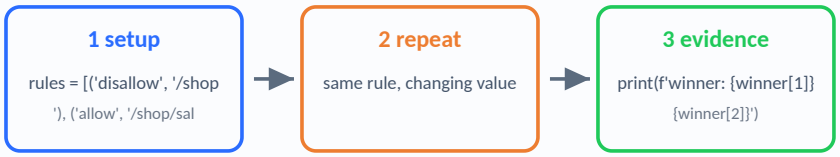
progress check

Reading target: trace overlapping rules to the winner is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `rules = [('disallow', '/shop'), ('allow', '/shop/sale'), ('disallow', '/shop/sale/expired')]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'winner: {winner[1]} {winner[2]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`rules = [('disallow', '/shop'), ('allow', '/shop/sale'), ('disallow', '/shop/sale/expired')]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`matches = [(len(p), a, p) for a, p in rules if path.startswith(p)]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'winner: {winner[1]} {winner[2]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `matches = [(len(p), a, p) for a, p in rules if path.startswith(p)]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Trace Overlapping Rules to the Winner: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: rules, disallow, shop, allow, sale. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: rules = [('disallow', '/shop'), ('allow', '/shop/sale'), ('disallow', '... | Core: matches = [(len(p), a, p) for a, p in rules if path.startswith(p)] | Evidence: print(f'winner: {winner[1]} {winner[2]}')

Reader to Writer Bridge

1. Read: explain trace overlapping rules to the winner in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 57.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 59 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'winner: {winner[1]} {winner[2]}')`. Then compare punctuation, spacing, and order.
2. Explain: point to `rules = [('disallow', '/shop'), ('allow', '/shop/sale'), ('disallow', '/shop/sale/expired')]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of trace overlapping rules to the winner and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Collecting every matching rule with its length, sorting by length (Allow breaking ties), and taking the last element exposes exactly why the deepest Disallow wins.

```
rules = [('disallow', '/shop'), ('allow', '/shop/sale'), ('disallow',
'/shop/sale/expired')]
path = '/shop/sale/expired/item'
matches = [(len(p), a, p) for a, p in rules if path.startswith(p)]
matches.sort(key=lambda x: (x[0], x[1] == 'allow'))
winner = matches[-1]
print('matching rules (by length):')
for length, action, pat in matches:
    print(f'  {action} {pat} (len {length})')
print(f'winner: {winner[1]} {winner[2]}')
```

```
output
matching rules (by length):
  disallow /shop (len 5)
  allow /shop/sale (len 10)
  disallow /shop/sale/expired (len 18)
winner: disallow /shop/sale/expired
```

How it works:

1. rules has a /shop Disallow, a /shop/sale Allow, and a deeper /shop/sale/expired Disallow.
2. matches collects (len(p), action, pattern) for every rule the path starts with.
3. Sorting by length then by the Allow flag puts the strongest match last.
4. The code prints each matching rule with its length, then the winner: disallow /shop/sale/expired.

Try it:

1. Add an even deeper Allow and watch it move to the end of the ranking
2. Print the path itself in the header so multi-path output stays labelled
3. Handle the case of no matching rules by reporting a default-allow verdict

4. Add an equal-length Allow to force a tie and watch it win.

⚠ Common Mistakes

- Sorting only by length without the Allow tiebreaker makes equal-length winners arbitrary.
- Taking `matches[0]` returns the shortest match, the opposite of longest-match.
- Forgetting that an empty match list would make `matches[-1]` raise IndexError.

Skills gained rule tracing | tuple sort | precedence visibility

→ Where You'll Use This: An explainable trace of competing rules is exactly what robots-testing tools surface so an SEO can see why a URL was blocked, not just that it was.

⚡ Mental Model

It is like showing the bracket before the final: you see every contender ranked by reach, so the champion at the bottom of the list is no surprise.

Project 59 Allow Everything When No Group Applies

💡 The Idea Behind It

If a bot matches no named group and there is no wildcard group either, it has zero applicable rules and may crawl the whole site. This project parses a file containing only a Googlebot block, then resolves rules for Bingbot, which is absent. With no wildcard fallback, rules is empty and the bot is free to crawl. This edge case trips parsers that assume a star always exists.

What You'll Build & Learn

Show a bot with no named group and no wildcard ends up with zero rules and full crawl access.

- no group means no rules
- absent wildcard grants full access
- do not assume a star block exists

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 58: trace overlapping rules to the winner. Project 60 changes the surface problem while preserving the underlying method.

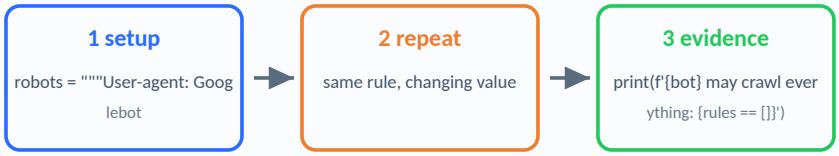
progress check

Reading target: allow everything when no group applies is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `robots = """User-agent: Googlebot` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{bot} may crawl everything: {rules == []}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: Googlebot`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{bot} may crawl everything: {rules == []}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Allow Everything When No Group Applies: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Googlebot, Disallow. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """User-agent: Googlebot | Core: if not line: | Evidence: print(f'{bot} may crawl everything: {rules == []}')

🛠️ Reader to Writer Bridge

1. Read: explain allow everything when no group applies in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 58.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 60 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{bot} may crawl everything: {rules == []}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `robots = """User-agent: Googlebot` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of allow everything when no group applies and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

With only a `Googlebot` block present, Bingbot finds neither its own group nor a `*` fallback, so its rule list is empty and nothing restricts it.

```
robots = """User-agent: Googlebot
Disallow: /no-google
"""
groups, agent = {}, None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    f, _, v = line.partition(':')
    f, v = f.strip().lower(), v.strip()
    if f == 'user-agent':
        agent = v; groups.setdefault(agent, [])
    elif agent:
        groups[agent].append((f, v))
bot = 'Bingbot'
if bot in groups:
    rules = groups[bot]
elif '*' in groups:
    rules = groups['*']
else:
    rules = []
print(f'{bot} applicable rules: {rules}')
print(f'{bot} may crawl everything: {rules == []}')
```

```
output
Bingbot applicable rules: []
Bingbot may crawl everything: True
```

How it works:

1. The robots string defines only a Googlebot block.
2. Parsing fills groups with a single Googlebot key.
3. For Bingbot, the code checks its own key, then the wildcard, then falls to an empty list.
4. rules is empty, so it prints that Bingbot may crawl everything as True.

Try it:

1. Add a wildcard block and confirm Bingbot now inherits its rules instead of crawling freely
2. Test Googlebot and verify it still uses its own named block
3. Print which branch supplied the rules so the fallback path is explicit
4. Add a `User-agent: *` block and watch Bingbot inherit it.

⚠ Common Mistakes

- Defaulting to a block when no group matches contradicts robots.txt's allow-by-default model.
- Crashing on a missing `*` key instead of treating absence as no-rules.
- Assuming an empty rule list means blocked rather than fully allowed.

Skills gained empty-rules edge case | fallback chain | default crawl

→ Where You'll Use This: Handling the no-applicable-group case prevents a crawler or audit from crashing or wrongly blocking an unnamed bot when a file targets only specific crawlers.

Mental Model

No matching group is an unguarded gate: with neither your name nor an all-comers sign on it, you simply walk through unchallenged.

Project 60 Match the Most Specific User-Agent Token

The Idea Behind It

User-agent matching is case-insensitive and picks the longest matching name, so Googlebot-Image obeys its own block rather than the generic googlebot one. This project lowercases the bot, scans the non-wildcard keys for substrings of the bot name, and keeps the longest. Token specificity, not file order, decides the group, and an unmatched bot falls back to the wildcard.

What You'll Build & Learn

Select a bot's group by the longest user-agent token contained in its name, falling back to the wildcard.

- user-agent matching is case-insensitive
- the longest token name wins
- specificity beats file order

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 59: allow everything when no group applies. Project 61 changes the surface problem while preserving the underlying method.

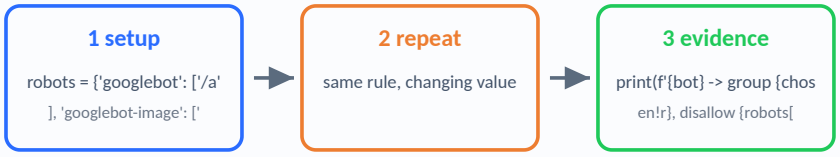
progress check

Reading target: match the most specific user-agent token is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `robots = {'googlebot': ['/a'], 'googlebot-image': ['/b'], '*': ['/c']}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{bot} -> group {chosen!r}, disallow {robots[chosen]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = {'googlebot': ['/a'], 'googlebot-image': ['/b'], '*': ['/c']}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if ua != '*' and ua in bot:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{bot} -> group {chosen!r}, disallow {robots[chosen]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `if ua != '*' and ua in bot:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Match the Most Specific User-Agent Token: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, googlebot, image, pick, bot. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: robots = {'googlebot': ['/a'], 'googlebot-image': ['/b'], '': *['/c']}* | *Core: if ua !=* '' and ua in bot: | Evidence: print(f'{bot} -> group {chosen!r}, disallow {robots[chosen]}')

🛠️ Reader to Writer Bridge

1. Read: explain match the most specific user-agent token in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 59.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 61 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{bot} -> group {chosen!r}, disallow {robots[chosen]}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `robots = {'googlebot': ['/a'], 'googlebot-image': ['/b'], '*': ['/c']}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of match the most specific user-agent token and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Lowercasing both sides and choosing the longest group name that is a substring of the bot string implements Google's most-specific-token rule, with `*` as the final fallback.

```
robots = {'googlebot': ['/a'], 'googlebot-image': ['/b'], '*': ['/c']}
def pick(bot):
    bot = bot.lower()
    best = None
    for ua in robots:
        if ua != '*' and ua in bot:
            if best is None or len(ua) > len(best):
                best = ua
    return best or '*'
for bot in ['Googlebot', 'Googlebot-Image', 'Bingbot']:
    chosen = pick(bot)
    print(f'{bot} -> group {chosen!r}, disallow {robots[chosen]}')
```

```
output
Googlebot -> group 'googlebot', disallow ['/a']
Googlebot-Image -> group 'googlebot-image', disallow ['/b']
Bingbot -> group '*', disallow ['/c']
```

How it works:

1. robots maps googlebot, googlebot-image, and wildcard keys to disallow lists.
2. pick lowercases the bot and scans non-wildcard keys for ones contained in the bot name.
3. It keeps the longest such match, defaulting to the wildcard when none is found.
4. Googlebot-Image picks its own longer token, while Bingbot matches nothing and lands on the wildcard.

Try it:

1. Add a googlebot-news key and confirm a news bot picks it over plain googlebot
2. Reorder the dictionary and verify the chosen group does not change
3. Reject a partial-word match like goog so only real token names count

4. Add a `googlebot-news` group and route a matching bot to it.

⚠ Common Mistakes

- Matching case-sensitively misses `Googlebot` against a lowercased `googlebot` group.
- Picking the first substring match instead of the longest sends `Googlebot-Image` to the wrong group.
- Letting `*` win as a substring of everything; it must be excluded from the specificity check.

Skills gained token specificity | case-insensitive match | substring scan

→ Where You'll Use This: Longest-token selection matters because Google runs many sub-bots like Googlebot-Image and Googlebot-News, each meant to honour its own most-specific block.

Mental Model

It is like name tags at a conference: Googlebot-Image wears the most specific badge it qualifies for, ignoring the broader googlebot table it could also sit at.

Project 61 Match a Mid-Path Wildcard and Query String

The Idea Behind It

A pattern like /products/*?sort= blocks any product URL carrying a sort parameter, using an asterisk in the middle of the path. This project compiles the wildcard pattern to a regex and tests several URLs. The mid-pattern star absorbs whatever path segment sits between /products/ and the query marker, which is exactly how robots.txt targets faceted-navigation URLs.

What You'll Build & Learn

Match URLs with a mid-path wildcard and a query fragment to target parameterised product pages.

- wildcards can sit mid-pattern
- a star spans path and query
- targets faceted navigation

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 60: match the most specific user-agent token. Project 62 changes the surface problem while preserving the underlying method.

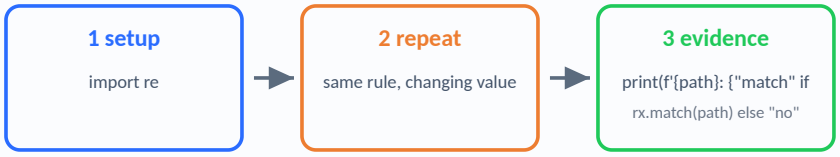
progress check

Reading target: match a mid-path wildcard and query string is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"match" if rx.match(path) else "no"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"match" if rx.match(path) else "no"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Match a Mid-Path Wildcard and Query String: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, to_rx, pat, rx, ch. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: rx.append('.' *if ch ==* '' else ('$' if ch == '$' else re.escape(ch))) | Evidence: print(f'{path}: {"match" if rx.match(path) else "no"}')

🛠️ Reader to Writer Bridge

1. Read: explain match a mid-path wildcard and query string in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 60.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 62 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"match" if rx.match(path) else "no"}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of match a mid-path wildcard and query string and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The `*` expands to `.*`, so `/products/` followed by anything and then `?sort=` matches faceted URLs while leaving the plain `/products/shoes` page crawlable.

```
import re
def to_rx(pat):
    rx = ['^']
    for ch in pat:
        rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))
    return re.compile(''.join(rx))
pattern = '/products/*?sort='
rx = to_rx(pattern)
paths = ['/products/shoes?sort=price', '/products/a/b?sort=name',
'/products/shoes', '/products/?sort=x']
for path in paths:
    print(f'{path}: {"match" if rx.match(path) else "no"}')
```

```
output
/products/shoes?sort=price: match
/products/a/b?sort=name: match
/products/shoes: no
/products/?sort=x: match
```

How it works:

1. to_rx compiles a pattern, mapping star to .* and dollar to $.
2. The pattern /products/*?sort= compiles to a regex requiring that literal prefix, any text, then ?sort=.
3. /products/shoes?sort=price and /products/a/b?sort=name match because .* spans the middle.
4. /products/shoes lacks the query fragment, so it does not match.

Try it:

1. Add a pattern catching ?color= as well and test a multi-parameter URL
2. Anchor the query so only sort exactly at the start of the query string matches
3. Confirm a page with sort in the path but not the query is correctly excluded
4. Add `?sort=` to `/products/shoes` and watch it start matching.

⚠ Common Mistakes

- Forgetting to escape the literal `?` lets it act as a regex 'optional' quantifier and over-match.
- Expecting `/products/shoes` to match; without `?sort=` it correctly does not.
- Assuming `*` stops at a `/`; it greedily crosses slashes, matching `/a/b` too.

Skills gained mid-path wildcard | query matching | faceted nav

→ Where You'll Use This: Mid-path wildcard patterns are the standard robots tool for pruning crawl waste from faceted navigation and sort parameters on large catalogue sites.

Mental Model

The mid-path star is a wildcard in the middle of a phone number: the prefix and the suffix are fixed, but any digits can fill the gap between them.

Project 62 Collapse Repeated Wildcards Before Matching

The Idea Behind It

Some robots.txt files write a double asterisk by mistake, but consecutive wildcards mean exactly the same as a single one. This project normalises runs of asterisks to one with a regex substitution before compiling the pattern, so /a/**/*.js collapses to /a//*.js. Normalising keeps the generated regex tidy and the matching behaviour predictable instead of letting redundant stars confuse it.

What You'll Build & Learn

Collapse consecutive wildcards into one before compiling so redundant asterisks behave predictably.

- double stars equal a single star
- normalise before compiling
- redundant wildcards are harmless

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 61: match a mid-path wildcard and query string. Project 63 changes the surface problem while preserving the underlying method.

progress check

Reading target: collapse repeated wildcards before matching is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path}: {"match" if rx.match(path) else "no"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {"match" if rx.match(path) else "no"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Collapse Repeated Wildcards Before Matching: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, normalize, pat, sub, to_rx. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: rx.append('.' *if ch ==* '' else ('$' if ch == '$' else re.escape(ch))) | Evidence: print(f'{path}: {"match" if rx.match(path) else "no"}')

🛠️ Reader to Writer Bridge

1. Read: explain collapse repeated wildcards before matching in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 61.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 63 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {"match" if rx.match(path) else "no"}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of collapse repeated wildcards before matching and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A regex pre-pass squeezes `*+` down to one `*`, so `/a/**/*.js` becomes `/a/*/*.js`, which still requires a directory segment between the two wildcards.

```
import re
def normalize(pat):
    return re.sub(r'\*+', '*', pat)
def to_rx(pat):
    pat = normalize(pat)
    rx = ['^']
    for ch in pat:
        rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))
    return re.compile(''.join(rx)), pat
raw = '/a/**/*.js'
rx, norm = to_rx(raw)
print(f'normalized {raw!r} -> {norm!r}')
for path in ['/a/b/c/x.js', '/a/x.js', '/b/x.js']:
    print(f'{path}: {"match" if rx.match(path) else "no"}')
```

```
output
normalized '/a/**/*.js' -> '/a/*/*.js'
/a/b/c/x.js: match
/a/x.js: no
/b/x.js: no
```

How it works:

1. normalize uses re.sub to replace any run of asterisks with a single asterisk.
2. to_rx normalises first, then compiles the cleaned pattern to an anchored regex.
3. /a/**/*.js normalises to /a/*/*.js and the code prints that transformation.
4. /a/b/c/x.js matches the two-star form, while /a/x.js lacks a middle segment and does not.

Try it:

1. Feed a triple-asterisk pattern and confirm it collapses to a single star
2. Print the regex both before and after normalisation to compare them
3. Decide whether to also trim a leading wildcard and test the effect
4. Feed `/a/***/x` and confirm it collapses to `/a/*/x`.

⚠ Common Mistakes

- Skipping normalization makes the generated regex have `.*.*`, which is correct but harder to read and reason about.
- Expecting `/a/x.js` to match; the pattern needs a segment between the wildcards, so it does not.
- Collapsing wildcards inside `re.escape`d text would corrupt the pattern; normalize before escaping.

Skills gained wildcard normalisation | re.sub | pattern cleanup

→ Where You'll Use This: Wildcard normalisation hardens a robots parser against the malformed patterns hand-written files contain, keeping verdicts stable regardless of redundant asterisks.

Mental Model

Two asterisks in a row are like saying anything anything: it adds no meaning, so normalising trims the stutter down to one clean wildcard.

Project 63 Block Query Strings but Keep One Path

The Idea Behind It

A Disallow: /*? rule hides every URL carrying a query string, while a specific Allow: /search keeps the search page crawlable. This project compiles both patterns, then resolves each URL by effective length with an Allow tie-break. The pairing is how sites prune crawl waste from parameter URLs while still letting a deliberately useful query-bearing page through.

What You'll Build & Learn

Block all query-string URLs with a wildcard Disallow while an Allow keeps one path crawlable.

- a /*? rule blocks all query URLs
- a specific Allow rescues one path
- prunes parameter crawl waste

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 62: collapse repeated wildcards before matching. Project 64 changes the surface problem while preserving the underlying method.

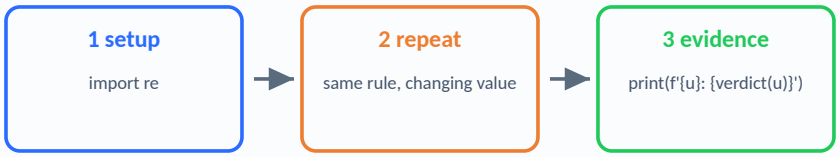
progress check

Reading target: block query strings but keep one path is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{u}: {verdict(u)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{u}: {verdict(u)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Block Query Strings but Keep One Path: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, to_rx, pat, rx, ch. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: rx.append('.' *if ch ==* '' else ('$' if ch == '$' else re.escape(ch))) | Evidence: print(f'{u}: {verdict(u)}')

🛠️ Reader to Writer Bridge

1. Read: explain block query strings but keep one path in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 62.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 64 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{u}: {verdict(u)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of block query strings but keep one path and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The wildcard `/*?` matches any path containing a `?`, blocking parameterised URLs, but the longer effective-length `/search` Allow rescues the search page even when it carries a query.

```
import re
def to_rx(pat):
    rx = ['^']
    for ch in pat:
        rx.append('.*' if ch == '*' else ('$' if ch == '$' else re.escape(ch)))
    return re.compile(''.join(rx))
rules = [('disallow', '/*?'), ('allow', '/search')]
def verdict(path):
    best_len, action = -1, 'allow'
    for a, pat in rules:
        if to_rx(pat).match(path):
            eff = len(pat.replace('*', ''))
            if eff > best_len or (eff == best_len and a == 'allow'):
                best_len, action = eff, a
    return action
urls = ['/page', '/page?ref=ad', '/search?q=x', '/list?sort=1']
for u in urls:
    print(f'{u}: {verdict(u)}')
```

```
output
/page: allow
/page?ref=ad: disallow
/search?q=x: allow
/list?sort=1: disallow
```

How it works:

1. rules pairs a Disallow /*? with an Allow /search.
2. verdict compiles each pattern, and for matches computes effective length with stars removed.
3. The longer effective length wins, with Allow favoured on a tie.
4. /page?ref=ad is disallowed by the query rule, but /search?q=x is allowed because /search matches longer.

Try it:

1. Add a second Allow for /filter and confirm it survives the query block
2. Tighten the Disallow to only block ?sort and retest the parameter URLs
3. Print the effective length of each matching rule for clarity

4. Add `/list` to an Allow and watch `/list?sort=1` open up.

⚠ Common Mistakes

- Forgetting effective-length precedence lets the short `/*?` block override the longer `/search` Allow.
- Escaping the `?` wrong makes the wildcard rule match the empty query and over-block.
- Treating `/page` (no query) as blocked; the `/*?` rule needs a `?` to match.

Skills gained query-string blocking | Allow exception | effective length

→ Where You'll Use This: The block-all-queries-with-one-exception pattern is a frequent crawl-budget tactic, hiding endless parameter permutations while protecting a key dynamic page.

Mental Model

Disallow: /*? is a blanket ban on anything wearing a query string, and the Allow: /search is the one VIP wristband that gets a parameterised page past the rope.

Project 64 Compare Percent-Decoded Paths in Rules

The Idea Behind It

A crawler compares robots rules against the percent-decoded path, so %20 and a literal space name the same target. This project decodes percent-escapes with a regex substitution that maps each hex pair to its character, then runs the prefix test on the decoded path. Decoding before matching prevents an encoded URL from sneaking past a Disallow that uses the plain character.

What You'll Build & Learn

Percent-decode requested paths before matching so encoded URLs cannot slip past a Disallow rule.

- rules match the decoded path
- %20 equals a literal space
- decode before comparing

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 63: block query strings but keep one path. Project 65 changes the surface problem while preserving the underlying method.

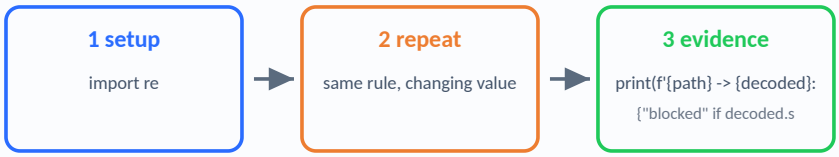
progress check

Reading target: compare percent-decoded paths in rules is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow) else "crawl"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow) else "crawl"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow) else "crawl"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE

```
print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow) else "crawl"}')
```

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Compare Percent-Decoded Paths in Rules: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, pct_decode, sub, Fa, lambda. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow)... | Evidence: print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow)...

Reader to Writer Bridge

1. Read: explain compare percent-decoded paths in rules in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 63.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 65 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow) else "crawl"}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compare percent-decoded paths in rules and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A small regex replaces each `%XX` escape with its character, so both the encoded and literal forms normalize to `/private folder` and match the same Disallow rule.

```
import re
def pct_decode(s):
    return re.sub(r'%([0-9A-Fa-f]{2})', lambda m: chr(int(m.group(1), 16)), s)
disallow = '/private folder'
requested = ['/private%20folder/x', '/private folder/y', '/public']
for path in requested:
    decoded = pct_decode(path)
    print(f'{path} -> {decoded}: {"blocked" if decoded.startswith(disallow)
else "crawl"}')
```

```
output
/private%20folder/x -> /private folder/x: blocked
/private folder/y -> /private folder/y: blocked
/public -> /public: crawl
```

How it works:

1. pct_decode uses re.sub to replace each %XX escape with chr of the parsed hex value.
2. disallow is /private folder with a literal space.
3. Each requested path is decoded, so /private%20folder becomes /private folder.
4. Both the encoded and literal-space paths block after decoding, while /public crawls.

Try it:

1. Add a path using %2F for a slash and decide how decoding should treat it
2. Decode the disallow value too so encoded rules are handled symmetrically
3. Handle a malformed escape like %2 gracefully instead of raising an error
4. Add `/private%2Ffolder` and note `%2F` should stay encoded.

⚠ Common Mistakes

- Matching the raw encoded path lets `/private%20folder` slip past a rule written with a space.
- Decoding `%2F` to `/` is unsafe; real parsers keep `%2F` encoded because it is a path-segment delimiter.
- A malformed `%2` with no following hex digit would be left untouched, which the regex handles by not matching.

Skills gained percent decoding | re.sub callback | normalised matching

→ Where You'll Use This: Percent-decoding before matching closes a real evasion gap, ensuring an audit treats encoded and plain URL forms identically as crawlers actually do.

⚡ Mental Model

Percent-encoding is a disguise; decoding first is the bouncer wiping off the costume so an escaped path cannot stroll past a rule written in plain characters.

Project 65 Catch Rules Above the First User-Agent

💡 The Idea Behind It

Allow and Disallow lines that appear before any User-agent line belong to no group and are ignored by crawlers. This project tracks whether a user-agent has been seen yet and, for each rule line that arrives before one, records an issue naming the line number and text. Catching these orphaned rules explains why a directive that should apply quietly does nothing.

What You'll Build & Learn

Flag Allow or Disallow lines that appear before the first User-agent and therefore belong to no group.

- rules need a User-agent above them
- orphaned lines are ignored
- report by line number

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 64: compare percent-decoded paths in rules. Project 66 changes the surface problem while preserving the underlying method.

progress check

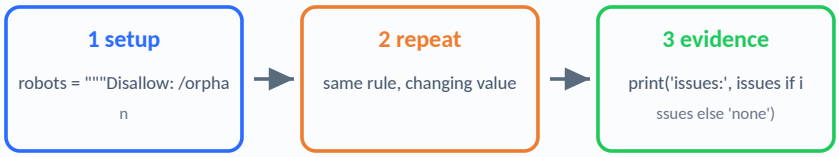
Reading target: catch rules above the first user-agent is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `robots = """Disallow: /orphan` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print('issues:', issues if issues else 'none')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """Disallow: /orphan`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('issues:', issues if issues else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Catch Rules Above the First User-Agent: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, Disallow, orphan, User, agent. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: robots = """Disallow: /orphan | Core: if not line: | Evidence: print('issues:', issues if issues else 'none')

🛠️ Reader to Writer Bridge

1. Read: explain catch rules above the first user-agent in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 64.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 66 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('issues:', issues if issues else 'none')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `robots = """Disallow: /orphan` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of catch rules above the first user-agent and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Tracking whether a `User-agent` line has been seen lets the parser flag any Allow/Disallow that precedes the first group as orphaned and inert.

```
robots = """Disallow: /orphan
User-agent: *
Disallow: /admin
Allow: /admin/help
"""
issues = []
agent = None
for n, line in enumerate(robots.splitlines(), 1):
    line = line.strip()
    if not line:
        continue
    f = line.partition(':')[0].strip().lower()
    if f == 'user-agent':
        agent = True
    elif f in ('allow', 'disallow') and not agent:
        issues.append(f'line {n}: {line!r} has no User-agent above it')
print('issues:', issues if issues else 'none')
```

```
output
issues: ["line 1: 'Disallow: /orphan' has no User-agent above it"]
```

How it works:

1. The robots string opens with a Disallow before any User-agent line.
2. agent starts None and enumerate provides each line's number.
3. A user-agent line sets agent to True; a rule line while agent is still None records an issue.
4. Line 1's orphaned Disallow: /orphan is reported, while the later rules under the wildcard are fine.

Try it:

1. Add a second orphaned Allow and confirm both are reported with their line numbers
2. Treat a Sitemap above the first group as valid since it is group-independent
3. Suggest moving each orphan under a User-agent line in the warning text
4. Move the orphan rule below the `User-agent` line and confirm the issue clears.

⚠ Common Mistakes

- Resetting the `agent` flag on blank lines wrongly orphans valid rules after a block break.
- Reporting line numbers off by one by not using `enumerate(..., 1)`.
- Treating a `Sitemap` line before the first group as orphaned; sitemaps are global and fine there.

Skills gained orphan detection | enumerate | line-number reporting

→ Where You'll Use This: Detecting orphaned rules is a useful linter check, explaining the confusing case where a correctly spelled directive has no effect because it sits above any group header.

Mental Model

An orphaned rule is a sentence before the chapter heading: it sits on the page but belongs to no section, so the reader skips right over it.

Project 66 Resolve Per-Bot Crawl-Delay Values

The Idea Behind It

Different bots can be given different Crawl-delay values, and a missing one falls back to the wildcard group's delay. This project parses crawl-delay per agent into a dictionary, then resolves the effective delay for each bot using get with the wildcard as the fallback. Computing the per-bot delay drives a polite scheduler that throttles aggressive crawlers harder than well-behaved ones.

What You'll Build & Learn

Resolve each bot's effective Crawl-delay, falling back to the wildcard group's value when unset.

- bots can have distinct crawl-delays
- missing delay falls to the wildcard
- resolve per-bot for scheduling

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 65: catch rules above the first user-agent. Project 67 changes the surface problem while preserving the underlying method.

progress check

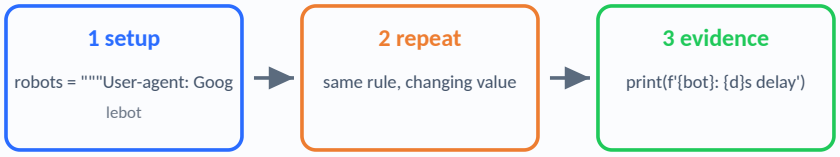
Reading target: resolve per-bot crawl-delay values is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `robots = """User-agent: Googlebot` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{bot}: {d}s delay')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`robots = """User-agent: Googlebot`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{bot}: {d}s delay')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Resolve Per-Bot Crawl-Delay Values: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: robots, User, agent, Googlebot, Crawl. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: robots = """User-agent: Googlebot | Core: if not line: | Evidence: print(f'{bot}: {d}s delay')

🛠️ Reader to Writer Bridge

1. Read: explain resolve per-bot crawl-delay values in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 65.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 67 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{bot}: {d}s delay')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `robots = """User-agent: Googlebot` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of resolve per-bot crawl-delay values and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Storing each group's Crawl-delay keyed by bot lets a lookup with a `*` fallback give every bot its effective wait, including the unlisted Yandex inheriting the wildcard's 5 seconds.

```
robots = """User-agent: Googlebot
Crawl-delay: 0

User-agent: BadBot
Crawl-delay: 30

User-agent: *
Crawl-delay: 5
"""
delays, agent = {}, None
for line in robots.splitlines():
    line = line.strip()
    if not line:
        continue
    f, _, v = line.partition(':')
    f, v = f.strip().lower(), v.strip()
    if f == 'user-agent':
        agent = v
    elif f == 'crawl-delay' and agent:
        delays[agent] = int(v)
for bot in ['Googlebot', 'BadBot', 'Yandex']:
    d = delays.get(bot, delays.get('*'))
    print(f'{bot}: {d}s delay')
```

```
output
Googlebot: 0s delay
BadBot: 30s delay
Yandex: 5s delay
```

How it works:

1. The robots string sets crawl-delays of 0 for Googlebot, 30 for BadBot, and 5 for the wildcard.
2. The parser tracks the current agent and stores each crawl-delay in the delays dictionary.
3. For each queried bot, delays.get(bot, delays.get('*')) returns its own delay or the fallback.
4. Yandex has no entry, so it inherits the wildcard's 5-second delay.

Try it:

1. Add Bingbot with its own delay and confirm it overrides the wildcard
2. Convert each resolved delay into a fetches-per-minute budget like an earlier project

3. Handle a robots file with no wildcard delay and report None gracefully
4. Remove the `*` Crawl-delay and decide what an unlisted bot should get.

⚠ Common Mistakes

- Falling back to a hardcoded default instead of the `*` group's delay ignores the file's own catch-all.
- Storing the delay as a string makes later arithmetic fail.
- Letting a Crawl-delay outside any group attach to the wrong bot.

Skills gained per-bot delay | dict fallback | scheduler input

→ Where You'll Use This: Per-bot crawl-delay resolution feeds a polite scheduler, letting a crawler obey a generous delay for friendly bots and a harsh one for resource-hungry scrapers.

Mental Model

It is a speed-limit sign per vehicle type: named bots read their own posted limit, and anything unlisted obeys the default sign at the entrance.

Project 67 Combine Meta and Header Robots Directives

The Idea Behind It

A page can carry indexing directives in both its meta robots tag and the X-Robots-Tag header, and the most restrictive wins. This project parses each source into a directive set, unions them, and checks for noindex in the combined set. A header noindex therefore overrides a meta tag that says index, showing how the two robots channels combine into one effective decision.

What You'll Build & Learn

Combine meta robots and X-Robots-Tag directives so the most restrictive instruction decides indexability.

- meta and header directives combine
- the most restrictive wins
- union the two directive sets

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 66: resolve per-bot crawl-delay values. Project 68 changes the surface problem while preserving the underlying method.

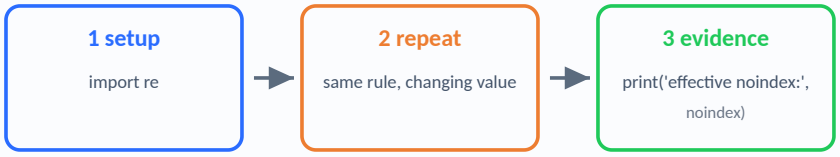
progress check

Reading target: combine meta and header robots directives is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print('effective noindex:', noindex)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`return {d.strip().lower() for d in text.split(',') if d.strip()}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('effective noindex:', noindex)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return {d.strip().lower() for d in text.split(',') if d.strip()}`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Combine Meta and Header Robots Directives: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, html, meta, name, robots. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: return {d.strip().lower() for d in text.split(',') if d.strip()} | Evidence: print('effective noindex:', noindex)

🛠️ Reader to Writer Bridge

1. Read: explain combine meta and header robots directives in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 66.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 68 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('effective noindex:', noindex)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of combine meta and header robots directives and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Unioning the meta and header directive sets and testing for `noindex` enforces the rule that any source asking for noindex wins, even against an explicit `index`.

```
import re
html = '<meta name="robots" content="index, follow">'
headers = {'X-Robots-Tag': 'noindex'}
def parse(text):
    return {d.strip().lower() for d in text.split(',') if d.strip()}
meta = set()
m = re.search(r'content=["\']([^"\']+)["\']', html)
if m:
    meta = parse(m.group(1))
hdr = parse(headers.get('X-Robots-Tag', ''))
combined = meta | hdr
noindex = 'noindex' in combined
print('meta:', sorted(meta))
print('header:', sorted(hdr))
print('effective noindex:', noindex)
```

```
output
meta: ['follow', 'index']
header: ['noindex']
effective noindex: True
```

How it works:

1. The meta tag says index, follow while the X-Robots-Tag header says noindex.
2. parse splits a comma string into a lowercased set; meta is read via a content regex.
3. combined is the union of the meta and header sets.
4. Because noindex is present in the header, effective noindex prints True despite the meta tag's index.

Try it:

1. Add nofollow to the header and report the combined link-following status
2. Reverse the conflict so the meta says noindex and the header says index
3. Print which source supplied the winning noindex directive
4. Add `nofollow` to the header and report the effective follow state.

⚠ Common Mistakes

- Letting the meta `index` cancel the header `noindex`; restrictive directives are not undone.
- Comparing the two sources instead of unioning them misses directives that appear in only one.
- Not lowercasing one source so `NoIndex` and `noindex` fail to match.

Skills gained directive merging set union restrictive wins

→ **Where You'll Use This:** Combining both robots channels is essential for diagnosing pages that stay out of the index because a header silently overrides an apparently index-friendly meta tag.

⚡ Mental Model

Two memos arrive about the same page; the strictest instruction governs, so a header that says keep it out of the index trumps a page that asks to be listed.

Project 68 Scope an X-Robots-Tag to a Named Bot

💡 The Idea Behind It

An X-Robots-Tag value can start with a bot name, like googlebot: noindex, scoping that directive to one crawler. This project inspects each header value, and when its prefix ends in bot it applies the directive only if that name matches my_bot; unprefixed values apply to every bot. The result is the set of directives that actually bind for the chosen crawler.

What You'll Build & Learn

Apply only the X-Robots-Tag directives scoped to a named bot plus the unprefixed ones meant for all.

- a header can name a target bot
- unprefixed directives apply to all
- prefixed ones bind one crawler

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 67: combine meta and header robots directives. Project 69 changes the surface problem while preserving the underlying method.

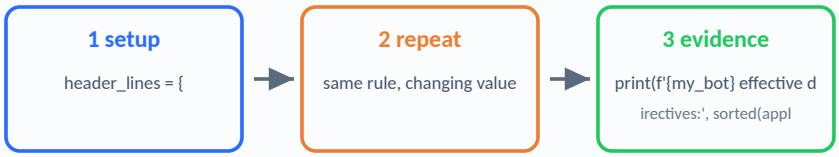
progress check

Reading target: scope an x-robots-tag to a named bot is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `header_lines = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{my_bot} effective directives:', sorted(applied))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`header_lines = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if sep and head.strip().lower().endswith('bot'):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{my_bot} effective directives:', sorted(applied))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if sep and head.strip().lower().endswith('bot'):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Scope an X-Robots-Tag to a Named Bot: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: header_lines, Robots, Tag, googlebot, noindex. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: header_lines = { | Core: if sep and head.strip().lower().endswith('bot'): | Evidence: print(f'{my_bot} effective directives:', sorted(applied))

🛠️ Reader to Writer Bridge

1. Read: explain scope an x-robots-tag to a named bot in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 67.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 69 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{my_bot} effective directives:', sorted(applied))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `header_lines = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scope an x-robots-tag to a named bot and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Splitting each header value on the first colon detects an optional bot prefix; a matching prefix or no prefix at all contributes its directives to the chosen bot's set.

```
header_lines = {
    'X-Robots-Tag': 'googlebot: noindex',
    'X-Robots-Tag-2': 'nosnippet',
}
my_bot = 'googlebot'
applied = set()
for value in header_lines.values():
    head, sep, rest = value.partition(':')
    if sep and head.strip().lower().endswith('bot'):
        if head.strip().lower() == my_bot:
            applied |= {d.strip() for d in rest.split(',')}
    else:
        applied |= {d.strip() for d in value.split(',')}
print(f'{my_bot} effective directives:', sorted(applied))
```

```
output
googlebot effective directives: ['noindex', 'nosnippet']
```

How it works:

1. header_lines holds a googlebot-scoped noindex and an unprefixed nosnippet.
2. Each value is partitioned on its first colon into head and rest.
3. When head ends in bot, the directive applies only if head equals my_bot; otherwise the whole value applies to all.
4. googlebot receives both noindex and nosnippet, the scoped one matching and the unscoped one applying universally.

Try it:

1. Add a bingbot-scoped directive and confirm googlebot does not pick it up
2. Handle a prefix that names a bot without the literal bot suffix
3. Resolve directives for a different my_bot and compare the applied sets
4. Change the prefix to `bingbot:` and watch `noindex` stop applying to Googlebot.

⚠ Common Mistakes

- Applying a `bingbot:`-prefixed directive to Googlebot by ignoring the prefix entirely.
- Splitting on every colon mangles values; partition on the first colon only.
- Treating an unprefixed value as bot-specific drops directives meant for all crawlers.

Skills gained scoped directives | header prefix parsing | bot targeting

→ Where You'll Use This: Bot-scoped X-Robots-Tag parsing matters when a server tailors indexing rules per crawler, ensuring an audit attributes each directive to the right bot rather than all of them.

Mental Model

A scoped header is mail with a name on the envelope: googlebot opens its addressed note plus every unaddressed all-staff memo, but not a note meant for bingbot.

Project 69 Run a Combined robots.txt Audit

The Idea Behind It

A real audit parses every directive in one pass: rules, sitemaps, and crawl-delay, then classifies a URL list for the active bot. This project strips comments, routes each field to its collection, compiles patterns to regex, and resolves verdicts by effective length with an Allow tie-break. That single pass is the skeleton of a robots.txt linter that reports the file's settings and per-URL outcomes.

What You'll Build & Learn

Parse rules, sitemaps, and crawl-delay in one pass, then classify a URL list for the active bot.

- one pass parses every directive
- route fields to collections
- classify URLs after parsing

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 68: scope an x-robots-tag to a named bot. Project 70 changes the surface problem while preserving the underlying method.

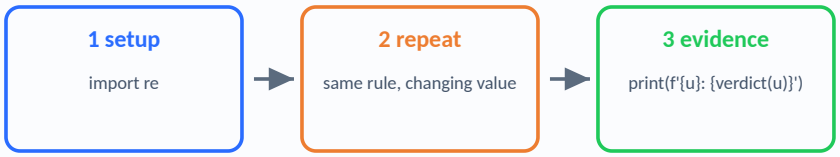
progress check

Reading target: run a combined robots.txt audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'{u}: {verdict(u)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
2. Apply the rule	`if not line:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{u}: {verdict(u)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not line:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Run a Combined robots.txt Audit: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, robots, User, agent, Disallow. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if not line: | Evidence: print(f'{u}: {verdict(u)}')

🛠️ Reader to Writer Bridge

1. Read: explain run a combined robots.txt audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 68.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 70 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{u}: {verdict(u)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of run a combined robots.txt audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

One scan collects rules, sitemaps, and the crawl-delay while a regex-backed `verdict` applies longest-match precedence, summarising both the file's metadata and its per-URL decisions.

```
import re
robots = """User-agent: *
Disallow: /admin
Disallow: /*.json$
Allow: /admin/public
Crawl-delay: 5
Sitemap: https://example.com/sitemap.xml
"""
rules, sitemaps, delay, agent = [], [], None, None
for line in robots.splitlines():
    line = line.split('#', 1)[0].strip()
    if not line:
        continue
    f, _, v = line.partition(':')
    f, v = f.strip().lower(), v.strip()
    if f == 'user-agent': agent = v
    elif f in ('allow', 'disallow') and agent: rules.append((f, v))
    elif f == 'crawl-delay': delay = int(v)
    elif f == 'sitemap': sitemaps.append(v)
def to_rx(p):
    return re.compile('^' + ''.join('.*' if c == '*' else ('$' if c == '$' else
re.escape(c)) for c in p))
def verdict(path):
    best, act = -1, 'allow'
    for a, p in rules:
        if to_rx(p).match(path):
            e = len(p.replace('*', '').replace('$', ''))
            if e > best or (e == best and a == 'allow'): best, act = e, a
    return act
urls = ['/home', '/admin', '/admin/public', '/data.json']
print('crawl-delay:', delay, '| sitemaps:', len(sitemaps))
for u in urls:
    print(f'{u}: {verdict(u)}')
```

```
output
crawl-delay: 5 | sitemaps: 1
/home: allow
/admin: disallow
```

```
/admin/public: allow
/data.json: disallow
```

How it works:

1. Each line has its comment stripped, then routes to rules, delay, or sitemaps by field.
2. to_rx compiles a pattern to an anchored regex handling star and dollar.
3. verdict keeps the longest effective-length match per URL, favouring Allow on ties.
4. It prints the crawl-delay and sitemap count, then classifies /admin as disallow and /admin/public as allow.

Try it:

1. Add the unknown-directive check from an earlier project into the same pass
2. Report the rule count and any orphaned rules alongside the existing summary
3. Accept a bot name argument so the audit selects the right group instead of the last agent
4. Add an `Allow: /data-public.json` and re-classify the JSON URLs.

⚠ Common Mistakes

- Letting a `Crawl-delay` or `Sitemap` line accidentally enter the Allow/Disallow rule list skews the verdicts.
- Stripping comments after splitting on the colon can drop a trailing `#` inside a value; strip first.
- Forgetting effective-length stripping makes the `/*.json$` rule mis-rank against `/admin/public`.

Skills gained single-pass parsing | robots linter | URL classification

→ Where You'll Use This: A one-pass audit is the working core of any robots.txt linter or testing tool an SEO runs to validate a file before or after a deploy.

Mental Model

This is the dashboard build: one sweep reads every gauge in the file at once, then runs your URLs through the wash and prints the verdicts.

Project 70 Report Crawl Coverage From a Policy

The Idea Behind It

The bottom line of a robots audit is coverage: of all the URLs you care about, how many can the bot actually reach. This project parses the rules, compiles patterns to regex, and resolves each URL to a boolean crawlable verdict, then splits the list into crawlable and blocked and prints a coverage ratio. It rolls parsing and matching into one crawlability score.

What You'll Build & Learn

Compute crawl coverage by classifying a URL list and reporting the crawlable fraction as a ratio.

- coverage measures reachable URLs
- split into crawlable and blocked
- report a crawlability ratio

Skill Roadmap

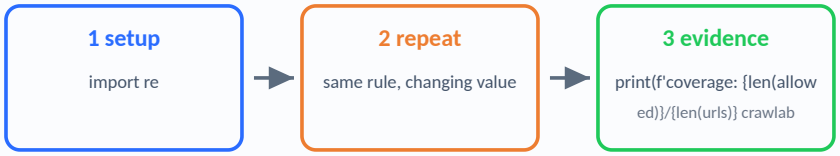
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 69: run a combined robots.txt audit. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: report crawl coverage from a policy is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as data -> transformation -> evidence, not as isolated library calls.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
- Evidence: `print(f'coverage: {len(allowed)}/{len(urls)} crawlable')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.

Move	Code to inspect	Why it matters
2. Apply the rule	`if f == 'user-agent': agent = v`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print(f'coverage: {len(allowed)}/{len(urls)} crawlable')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: robots.txt and Crawl Directives becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if f == 'user-agent': agent = v`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Report Crawl Coverage From a Policy: connect the surface task to the chapter skill, parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay.
Names	Watch these names first: re, robots, User, agent, Disallow. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: if f == 'user-agent': agent = v | Evidence: print(f'coverage: {len(allowed)}/{len(urls)} crawlable')

Reader to Writer Bridge

1. Read: explain report crawl coverage from a policy in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 69.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'coverage: {len(allowed)}/{len(urls)} crawlable')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of report crawl coverage from a policy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Parsing the rules once and running each URL through a longest-match `verdict` partitions the list into crawlable and blocked sets, and the ratio is the site's crawl coverage under this policy.

```
import re
robots = """User-agent: *
Disallow: /search
Disallow: /*.json$
Allow: /search/help
Sitemap: https://example.com/sitemap.xml
"""
rules, agent = [], None
for line in robots.splitlines():
    f, _, v = line.partition(':')
    f, v = f.strip().lower(), v.strip()
    if f == 'user-agent': agent = v
    elif f in ('allow', 'disallow') and agent: rules.append((f, v))
def to_rx(p):
    return re.compile('^' + ''.join('.*' if c == '*' else ('$' if c == '$' else
re.escape(c)) for c in p))
def verdict(path):
    best, act = -1, 'allow'
    for a, p in rules:
        if to_rx(p).match(path):
            e = len(p.replace('*', '').replace('$', ''))
```

```
            if e > best or (e == best and a == 'allow'): best, act = e, a
    return act == 'allow'
urls = ['/home', '/search', '/search/help', '/feed.json', '/about']
allowed = sorted(u for u in urls if verdict(u))
blocked = sorted(u for u in urls if not verdict(u))
print('crawlable:', allowed)
print('blocked:', blocked)
print(f'coverage: {len(allowed)}/{len(urls)} crawlable')
```

```
output
crawlable: ['/about', '/home', '/search/help']
blocked: ['/feed.json', '/search']
coverage: 3/5 crawlable
```

How it works:

1. Rules are parsed for the active agent, ignoring sitemaps for the verdict.
2. verdict compiles each pattern and returns whether the longest effective-length match is an allow.
3. The URL list is partitioned into sorted allowed and blocked lists by that verdict.
4. The code prints both lists and a coverage line reading 3 of 5 crawlable.

Try it:

1. Print the coverage as a percentage in addition to the raw ratio
2. Add more URLs including a .json file and confirm the end-anchored rule blocks it
3. Group blocked URLs by which rule blocked them for a richer report
4. Add `/search/secret` and confirm the `/search` Disallow blocks it.

⚠ Common Mistakes

- Calling `verdict` twice per URL (once for each list) is wasteful; compute it once and bucket the result.
- Forgetting that `/search/help` survives because its Allow out-ranks the `/search` Disallow.
- Sorting only one of the two lists makes the output partly nondeterministic.

Skills gained crawl coverage | URL partitioning | coverage ratio

→ Where You'll Use This: A coverage score is the headline metric of a crawlability report, telling an SEO at a glance how much of a target URL set a robots policy actually permits.

Mental Model

Coverage is the attendance count after the door policy runs: of everyone you invited, it tells you what share actually made it inside.

✅ Chapter Progress Checkpoint

What this chapter added: after 40 mini-projects, robots.txt and Crawl Directives should feel like a reusable Python move: parse and validate robots rules, user-agent matching, allow/disallow precedence and crawl-delay. *chapter checkpoint*

1 Read
Open any project from Projects 31-70 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 3 (XML Sitemaps and URL Discovery) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

6 questions · 4 categories · graded

1 Recall · 3 Meaning · 1 Structure · 1 Transfer

2.1.Q1 Recall · MEDIUM
Turn back to Project 42 (Treat an Empty Disallow as Allow-All). Without running it, what exactly does it print?

2.1.Q2 Meaning · MEDIUM
Turn back to Project 41 (Detect a Site-Wide Disallow Block). Without running it, what exactly does it print?

2.1.Q3 Structure · EASY
A robots.txt names a **`Googlebot`** block AND a `*` block. Which rules does Googlebot obey?

- A. Both blocks combined
- B. Only the `*` block
- C. Only its own `Googlebot` block
- D. Whichever block comes first

2.1.Q4 Transfer MEDIUM For path **`/public/page`**, the active group has **`Disallow: /`** and **`Allow: /public`**. What is the verdict and why?

- A Blocked — Disallow always wins
- B Crawl — `/public` (7 chars) is a longer match than `/` (1 char)
- C Blocked — the first rule wins
- D Crawl — Allow always wins

2.1.Q5 Meaning EASY What does a trailing **`0`** ?

- A Marks the end of the file
- B Means "match any single character"
- C Anchors the pattern to the end of the path, so it matches only paths that END there
- D Escapes the preceding character

2.1.Q6 Meaning EASY A path matches no Allow or Disallow rule in the bot's group. Is it crawlable?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 2 – robots.txt and Crawl Directives

2.1.Q1. `empty Disallow -> entire site allowed`

2.1.Q2. `disallow rules: ['/']` then `blocks entire site: True`

2.1.Q3. C a bot with its own named group obeys only that group and ignores the * block entirely.

2.1.Q4. B longest-match precedence: the 7-character Allow out-matches the 1-character Disallow.

2.1.Q5. C $ requires the path to end at that point, so `/a.pdf` matches but `/a.pdf?v=2` does not.

2.1.Q6. Yes — robots.txt is permissive by default, so any path the file never mentions is crawlable.

Chapter 3 · XML Sitemaps and URL Discovery

Build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.

Chapter Skill Map

What this chapter adds: In XML Sitemaps and URL Discovery, the focus is build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors. You practise it across 40 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples.

chapter arc

What it builds on: Chapter 3 assumes the reading habits from Chapter 2: robots.txt and Crawl Directives. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 4, HTTP Status Codes and Redirects, reuses this chapter's discipline with a new kind of problem.

1 **Projects 71-83**	2 **Projects 84-96**	3 **Projects 97-110**
Recognise the pattern and read the code slowly. name it first	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. own the move

The Idea Behind It

A sitemap is a machine-readable list of the URLs you want found, written as XML.

you tell the crawler where to look, instead of hoping it finds the links

In Chapter 1 the crawler discovered pages by walking the link graph — follow an edge, fetch a node, repeat. That works, but it is at the mercy of your internal linking: an orphan page with no inbound link is invisible. A sitemap is the second discovery channel. It is a file you publish that simply *lists the URLs you want crawled*, so the bot has a seed set that does not depend on link-following at all.

The format is plain XML. The top-level element is a `<urlset>`, and inside it sits one `<url>` entry per page. Each `<url>` must contain exactly one `<loc>` — the absolute URL — and may optionally carry `<lastmod>`, `<changefreq>` and `<priority>`. That is the entire grammar. Parsing one is the first move of any sitemap audit: read the urlset, pull each `<loc>`.

We show sitemaps here as namespace-free fixtures — bare `<urlset>` and `<url>` tags — so the parsing logic stays readable. A real sitemap declares the sitemaps.org namespace on the root element (an `xmlns` attribute holding an http URI). It changes how you address the tags but not the structure you are learning.

real file: <urlset xmlns="...sitemaps.org schema..."> — we omit it on purpose

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/</loc></url>'
       '<url><loc>https://example.com/blog</loc></url>'
       '<url><loc>https://example.com/about</loc></url>'
       '</urlset>')
```

```
root = ET.fromstring(xml)
for u in root.findall('url'):
    print(u.findtext('loc'))
```

```
output
https://example.com/
https://example.com/blog
https://example.com/about
```

That loop is the whole skeleton of a sitemap tool. Everything else in this chapter — counting URLs, validating dates, flagging out-of-range priorities, splitting oversized files — is a variation on *parse the urlset, then check each entry*. parse once, audit each <url>

One sitemap cannot grow forever. A single file is capped at 50,000 URLs and 50 MB uncompressed. Past either limit you split the URLs across several sitemaps and publish one `<sitemapindex>` that points to them — a sitemap *of sitemaps*, each child referenced by its own `<loc>`. Big sites are always a shallow tree: one index at the top, many urlsets below.

SITEMAP

An XML file you publish listing the URLs you want a search engine to crawl, giving the bot a discovery seed independent of internal links.

URLSET

The root `<urlset>` element of a normal sitemap; it contains one `<url>` entry per page.

LOC

The required child of each `<url>`: the page's absolute, fully-qualified URL (e.g. `https://example.com/blog`). A `<url>` with no `<loc>` is invalid.

LASTMOD

Optional date a page last changed, in W3C/ISO form like `2026-01-01` (or a full timestamp with a `T`). It hints to crawlers what is fresh.

CHANGEFREQ & PRIORITY

<changefreq> is one of seven keywords (always, hourly, daily, weekly, monthly, yearly, never); <priority> is a decimal 0.0–1.0 (default 0.5). Both are hints, not commands.

SITEMAPINDEX

A root <sitemapindex> element whose <sitemap> children each <loc> a child sitemap file — used to fan out past the 50,000-URL / 50 MB limit.

The grammar: urlset, url, loc

A sitemap is a tree with a fixed shape. The root is <urlset>. Each child is a <url>. Inside every <url> is exactly one <loc> (required) plus optional metadata. So the *number of URLs a sitemap declares* is simply the number of <url> elements — the simplest health metric there is, and the quantity a search console reports back to you:

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/</loc></url>'
       '<url><loc>https://example.com/blog</loc></url>'
       '<url><loc>https://example.com/shop</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
print('sitemap contains', len(root.findall('url')), 'URLs')
```

```
output
sitemap contains 3 URLs
```

Because <loc> is the one required field, the first validation any linter runs is *does every <url> actually have one?* A <url> carrying only a <lastmod> and no <loc> is a broken entry the crawler cannot use.

no loc, no URL, no point

The optional metadata: lastmod, changefreq, priority

Beyond <loc>, three optional tags add signal. <lastmod> is a date; Google expects ISO/W3C form like 2026-01-01, so slashes or a reversed day-month order are rejected. <changefreq> must be one of seven exact keywords. <priority> is a decimal from 0.0 to 1.0 and defaults to 0.5 when omitted — so a sitemap where every page claims 1.0 sends no signal at all. Each tag has its own validity rule, and an audit checks all three:

```
import xml.etree.ElementTree as ET
from datetime import datetime
ALLOWED = {'always','hourly','daily','weekly','monthly','yearly','never'}
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc>'
```

```
        '<lastmod>2026/02/15</lastmod>'
        '<changefreq>often</changefreq>'
        '<priority>1.5</priority></url>'
        '</urlset>')
root = ET.fromstring(xml)
u = root.find('url')
try:
    datetime.strptime(u.findtext('lastmod'), '%Y-%m-%d'); d = 'ok'
except ValueError:
    d = 'BAD DATE'
p = float(u.findtext('priority'))
print('lastmod:', d)
print('changefreq:', 'ok' if u.findtext('changefreq') in ALLOWED else 'INVALID')
print('priority:', 'ok' if 0.0 <= p <= 1.0 else 'OUT OF RANGE')
```

```
output
lastmod: BAD DATE
changefreq: INVALID
priority: OUT OF RANGE
```

All three fields in that entry are wrong — and notice `datetime.strptime` does more than a regex could: it would also reject `2026-02-30`, a date that *looks* valid but is not a real calendar day.

strptime knows February has 28 or 29 days

Locations must be absolute, same-host HTTPS

A `<loc>` is not a path — it must be a fully-qualified absolute URL on the host that serves the sitemap. Relative paths (`/about`), scheme-less strings (`example.com/x`), non-HTTPS schemes (`ftp://`) and cross-domain URLs are all rejected or ignored by crawlers. The cheapest sound check is "does it start with `https://`?":

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>/relative/path</loc></url>'
       '<url><loc>ftp://example.com/file.zip</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    print(loc, '->', 'ok' if loc.startswith('https://') else 'NOT ABSOLUTE HTTPS')
```

```
output
https://example.com/a -> ok
/relative/path -> NOT ABSOLUTE HTTPS
ftp://example.com/file.zip -> NOT ABSOLUTE HTTPS
```

One more well-formedness trap lives in the URL itself: a raw & in a query string breaks XML parsing and must be escaped to &. So `https://example.com/search?q=shoes&sort=price` has to be written `https://example.com/search?q=shoes&sort=price` inside the file.

The two hard limits and the sitemap index

A single sitemap may hold at most 50,000 URLs and weigh at most 50 MB uncompressed. Hit either and you must split the list into multiple files. The check on each side is just a measurement — count the entries, or measure the encoded byte length:

```
urls = [f'https://example.com/p/{i}' for i in range(50002)]
LIMIT = 50000
n = len(urls)
print(n, 'URLs (limit', LIMIT, ') ->', 'SPLIT NEEDED' if n > LIMIT else 'ok')
print('overflow:', n - LIMIT)
```

```
output
50002 URLs (limit 50000 ) -> SPLIT NEEDED
overflow: 2
```

When you split, you publish one <sitemapindex> — a sitemap whose children are *other sitemaps*. Each <sitemap> child carries its own <loc> (and optional <lastmod>). Reading the index is how a crawler fans out from one file to dozens; note that the index hides the real URL total, so to know your full declared coverage you sum each child's count:

```
import xml.etree.ElementTree as ET
xml = ('<sitemapindex>'
       '<sitemap><loc>https://example.com/sitemap-posts.xml</loc>'
       '<lastmod>2026-01-10</lastmod></sitemap>'
       '<sitemap><loc>https://example.com/sitemap-pages.xml</loc>'
       '<lastmod>2026-02-01</lastmod></sitemap>'
       '</sitemapindex>')
root = ET.fromstring(xml)
for sm in root.findall('sitemap'):
    print(sm.findtext('loc'), '->', sm.findtext('lastmod'))
print('child sitemaps:', len(root.findall('sitemap')))
```

```
output
https://example.com/sitemap-posts.xml -> 2026-01-10
https://example.com/sitemap-pages.xml -> 2026-02-01
child sitemaps: 2
```

Sitemaps drive URL discovery — and reveal gaps

The payoff is discovery. The set of <loc> values *is* the seed set the crawler can fetch without following a single link, which is exactly how you surface orphan pages. But the sitemap is most useful when you compare it to the

crawl. Set subtraction exposes two different problems: URLs you declared that were never reached, and URLs the crawler found that you forgot to declare.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '<url><loc>https://example.com/orphan</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
sitemap = {u.findtext('loc') for u in root.findall('url')}
crawled = {'https://example.com/a', 'https://example.com/b',
'https://example.com/extra'}
print('in sitemap, not crawled:', sorted(sitemap - crawled))
print('crawled, not in sitemap:', sorted(crawled - sitemap))
```

```
output
in sitemap, not crawled: ['https://example.com/orphan']
crawled, not in sitemap: ['https://example.com/extra']
```

A URL in the sitemap but never crawled may be orphaned or blocked; a crawled URL missing from the sitemap probably ought to be added. The sitemap is not a magic ranking lever — it is a *discovery and inventory* tool.

sitemap ∪ crawl tells you what you have; the difference tells you what to fix

Example 1

You parse a urlset and find an entry: <url><loc>https://example.com/sale</loc><lastmod>2026/03/01</lastmod><priority>0.9</priority><changefreq>weekly</changefreq></url>. Which fields, if any, are invalid?

SOLUTION

1. `<loc>` is an absolute `https://` URL on the host — valid.
2. `<lastmod>` is `2026/03/01` with slashes; the ISO form needs hyphens (`2026-03-01`), so this is invalid date format.
3. `<priority>` is 0.9, inside 0.0–1.0 — valid; `<changefreq>` is `weekly`, one of the seven keywords — valid.

Only `<lastmod>` is invalid: the date uses slashes instead of the required ISO hyphen form. The loc, priority and changefreq are all fine.

one bad tag does not invalidate the others, but a linter still flags it

Example 2

A site has 50,000 product URLs plus 12,000 blog URLs. Can they go in one sitemap? If not, how do you structure the files?

SOLUTION

1. Total URLs = 50,000 + 12,000 = 62,000, which exceeds the 50,000-URL cap for a single sitemap.
2. So you split: e.g. `sitemap-products.xml` (50,000) and `sitemap-blog.xml` (12,000), each under the limit.
3. You then publish one `<sitemapindex>` whose two `<sitemap>` children `<loc>` those files, so the crawler reads the index and fans out to both.

No — 62,000 > 50,000, so you split into child sitemaps and reference them from a single sitemap index. Summing the children (50,000 + 12,000) gives the true declared coverage of 62,000.

index at the top, urlsets below

Mental-Math Shortcut

List it, don't lose it. A sitemap is just "the URLs I want found, written down." Every audit reduces to: parse the `<urlset>`, and for each `<url>` ask is the `<loc>` an absolute same-host HTTPS URL, is the `<lastmod>` a real ISO date, is `<priority>` in 0.0–1.0, is `<changefreq>` one of the seven words — and does the whole file stay under 50,000 URLs and 50 MB. Past those caps, fan out with a `<sitemapindex>`.

⚠ Common Mistakes

- Putting relative paths or non-HTTPS / cross-domain URLs in <loc>. Locations must be absolute, fully-qualified URLs on the same host that serves the sitemap, or crawlers ignore them.
- Writing <lastmod> with slashes or reversed day-month order (2026/03/01, 15-03-2026). Google expects ISO/W3C form 2026-03-01; strptime also catches impossible dates like 2026-02-30.
- Letting a single sitemap exceed 50,000 URLs or 50 MB. Past either cap you must split the URLs and publish a <sitemapindex> pointing to the child files.
- Treating <priority> and <changefreq> as ranking levers. They are crawl hints only; setting every page to priority 1.0 conveys nothing, and a raw & in a URL must still be escaped to & or the XML breaks.

This chapter contains 40 hands-on mini-projects (Projects 71–110). Type each one out, run it, and experiment.

Project 71 Extract Every loc From a Urlset

The Idea Behind It

A sitemap is a urlset wrapper holding many url elements, and every url carries exactly one loc with the page address. This project loads a urlset string with ElementTree, walks each url node, and pulls the text inside its

loc child into a list. It proves that reading a sitemap is really just tree traversal: find the repeated element, read its child, collect the values.

What You'll Build & Learn

Parse a urlset string and print every URL declared inside its loc tags.

- ElementTree parses XML into a node tree
- findall gathers all repeated url elements
- findtext reads a child element's inner text

🧭 Skill Roadmap

Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 72 changes the surface problem while preserving the underlying method. *progress check*

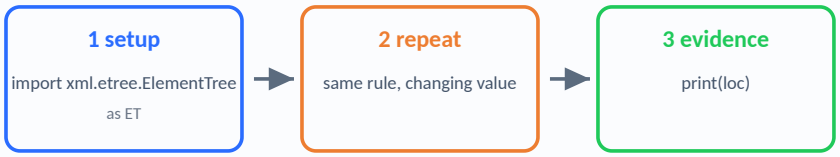
Reading target: extract every loc from a urlset is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(loc)` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`locs = [u.findtext('loc') for u in root.findall('url')]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(loc)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `locs = [u.findtext('loc') for u in root.findall('url')]`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Extract Every loc From a Urlset: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: locs = [u.findtext('loc') for u in root.findall('url')] | Evidence: print(loc)

🛠️ Reader to Writer Bridge

1. Read: explain extract every loc from a urlset in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 72 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(loc)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of extract every loc from a urlset and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`ET.fromstring` parses the XML string into an element tree, and `findall('url')` plus `findtext('loc')` reads each entry's location; real sitemaps add an `xmlns` attribute but this parsing logic is identical.

```
import xml.etree.ElementTree as ET
xml = '<urlset><url><loc>https://example.com/</loc></url><url>
<loc>https://example.com/blog</loc></url><url>
<loc>https://example.com/about</loc></url></urlset>'
root = ET.fromstring(xml)
locs = [u.findtext('loc') for u in root.findall('url')]
for loc in locs:
    print(loc)
```

```
output
https://example.com/
https://example.com/blog
https://example.com/about
```

How it works:

1. ET.fromstring turns the urlset string into a root element holding three url children.
2. root.findall('url') returns the list of url nodes to iterate.
3. The list comprehension calls u.findtext('loc') on each, extracting the URL string.
4. The for loop prints each loc on its own line.

Try it:

1. Add a fourth url element and confirm the new loc prints.
2. Collect the locs into a set instead of a list to drop accidental duplicates.
3. Print the total count of URLs after the loop finishes.
4. Add a fourth `<url>` and confirm the loop prints it too.

⚠ Common Mistakes

- Real sitemaps declare a default namespace, which would make `findall('url')` return nothing unless you prefix the tag; this fixture omits it on purpose.
- Using `root.findall('loc')` looks one level too shallow because `<loc>` is nested inside `<url>`.
- `find` returns only the first match, so use `findall` to get every URL.

Skills gained ElementTree parsing | findall | findtext | list comprehension

→ Where You'll Use This: Extracting the loc list is the opening move of any sitemap audit, feeding crawl comparisons, broken-link checks, and coverage reports downstream.

Mental Model

The urlset is a shipping manifest and each loc is one parcel's address: you read the manifest line by line to learn where everything goes.

Project 72 Count the URLs in a Sitemap

The Idea Behind It

Search consoles surface how many URLs a sitemap declares, and that figure is simply the number of url elements. This code parses a urlset, calls findall to gather every url node into a list, and reports len of that list inside an f-string. It is the smallest possible health metric and the foundation for the later check that no file exceeds fifty thousand entries.

What You'll Build & Learn

Count the url elements in a sitemap and print the total in a readable sentence.

- findall returns a list you can measure
- len gives the declared URL count
- Counting is the first sitemap health check

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 71: extract every loc from a urlset. Project 73 changes the surface problem while preserving the underlying method.

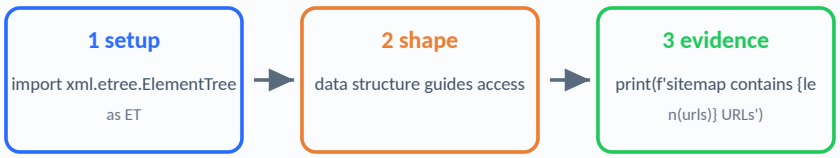
progress check

Reading target: count the urls in a sitemap is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The data structure is doing part of the thinking. Ask what the collection stores, how items are found, and what order or uniqueness it promises.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'sitemap contains {len(urls)} URLs')` exposes the result; match it against the output before changing the program. output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, shape applies the chapter idea, and evidence proves your trace against the output. visual trace

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`xml = '<urlset><url><loc>https://example.com/</loc></url><url><loc>https://example.com/blog</loc></url><url><loc>https://example.com/shop</loc></url></urlset>'`	This line depends on the chosen data shape. Ask how items are stored, found, or ordered. slow line
3. Check the evidence	`print(f'sitemap contains {len(urls)} URLs')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
xml = '<urlset><url><loc>https://example.com/</loc></url><url>
<loc>https://example.com/blog</loc></url><url><loc>https://example.com/shop</...
```

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Name the collection shape, then ask how an item is added, selected, transformed, or counted.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Count the URLs in a Sitemap: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: xml = '<urlset><url><loc>https://example.com/</loc></url><url><loc>http... | Evidence: print(f'sitemap contains {len(urls)} URLs')

Reader to Writer Bridge

1. Read: explain count the urls in a sitemap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 71.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 73 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'sitemap contains {len(urls)} URLs')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count the urls in a sitemap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`findall('url')` returns a list of every `<url>` element, so its length is the declared URL count for the sitemap.

```
import xml.etree.ElementTree as ET
xml = '<urlset><url><loc>https://example.com/</loc></url><url>
<loc>https://example.com/blog</loc></url><url>
<loc>https://example.com/shop</loc></url></urlset>'
root = ET.fromstring(xml)
urls = root.findall('url')
print(f'sitemap contains {len(urls)} URLs')
```

```
output
sitemap contains 3 URLs
```

How it works:

1. ET.fromstring parses the urlset into a root element.
2. root.findall('url') collects the three url nodes into a list named urls.
3. len(urls) yields the count of declared URLs.
4. The f-string prints sitemap contains 3 URLs.

Try it:

1. Warn when the count exceeds a threshold you choose, like 2.
2. Print the count of unique locs alongside the raw url total.
3. Wrap the count in a function that accepts any urlset string.
4. Add a `<url>` with no `<loc>` and confirm it still counts as a URL entry.

⚠ Common Mistakes

- Counting `<loc>` tags instead of `<url>` entries miscounts when a url block holds extra child tags.
- `len(root)` counts direct children which happens to match here but breaks if non-url tags appear.
- A real namespaced sitemap needs the prefixed tag name or the count comes back zero.

Skills gained `findall` `len` `f-string`

→ **Where You'll Use This:** URL counts feed directly into Search Console comparisons, where a mismatch between declared and indexed pages flags coverage problems worth investigating.

Mental Model

Counting url elements is like tallying chairs in a room before a meeting: a quick headcount tells you whether the space fits everyone.

Project 73 Flag URLs Missing a lastmod

The Idea Behind It

The lastmod tag is optional, yet it tells crawlers when a page changed, so auditors list the URLs that omit it. This project parses each url, reads both loc and lastmod with findtext, and substitutes a MISSING lastmod marker when findtext returns a falsy value. The conditional cleanly separates dated pages from undated ones so editors can backfill the gaps.

What You'll Build & Learn

Report each URL's lastmod value, flagging the entries that lack one.

- findtext returns None when a child is absent
- A truthiness test selects a fallback label
- lastmod signals page freshness to crawlers

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 72: count the urls in a sitemap. Project 74 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag urls missing a lastmod is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.

- Evidence: `print(f'{loc}: {status}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`status = lastmod if lastmod else 'MISSING lastmod'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: {status}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `status = lastmod if lastmod else 'MISSING lastmod'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag URLs Missing a lastmod: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: status = lastmod if lastmod else 'MISSING lastmod' | Evidence: print(f'{loc}: {status}')

🛠️ Reader to Writer Bridge

1. Read: explain flag urls missing a lastmod in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 72.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 74 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {status}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag urls missing a lastmod and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`findtext('lastmod')` returns `None` when the tag is absent, so a falsy check cleanly separates dated URLs from those needing a lastmod.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/</loc><lastmod>2026-01-01</lastmod>
</url>'
       '<url><loc>https://example.com/blog</loc></url>'
```

```
        '<url><loc>https://example.com/shop</loc><lastmod>2026-02-15</lastmod>
</url>'
        '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    lastmod = u.findtext('lastmod')
    status = lastmod if lastmod else 'MISSING lastmod'
    print(f'{loc}: {status}')
```

```
output
https://example.com/: 2026-01-01
https://example.com/blog: MISSING lastmod
https://example.com/shop: 2026-02-15
```

How it works:

1. The urlset holds three url nodes; the middle one has no lastmod child.
2. For each url, loc and lastmod are read via findtext.
3. status becomes the lastmod value if present, otherwise the string MISSING lastmod.
4. The f-string prints each loc followed by its status.

Try it:

1. Count how many URLs are missing a lastmod and print the total.
2. Collect only the missing locs into a separate list for export.
3. Replace the marker text with a sortable empty string and re-sort.
4. Add an empty `<lastmod></lastmod>` and see it flagged as missing.

⚠ Common Mistakes

- `findtext` returns `None` (not an empty string) for a missing tag, so `lastmod == ''` would never match.
- An empty `<lastmod></lastmod>` returns `''`, also falsy, which may or may not be what you want to flag.
- Treating a missing lastmod as an error rather than a warning; it is valid to omit it.

Skills gained findtext | conditional expression | truthiness

→ Where You'll Use This: Backfilling missing lastmod dates helps crawlers schedule recrawls efficiently, so this gap report drives a concrete content-hygiene task list.

Mental Model

Think of lastmod like a sell-by date: a missing one does not spoil the item, but stockers cannot tell what needs rotating to the front.

Project 74 Validate lastmod as an ISO Date

The Idea Behind It

Google expects lastmod in W3C/ISO form like 2026-01-01, and this project enforces that by trying to parse each value with a fixed strptime pattern. When the format has slashes or a reversed day-month order, strptime raises ValueError, which the try/except catches and labels BAD FORMAT. The technique turns a vague formatting rule into a hard pass/fail per URL.

What You'll Build & Learn

Validate every lastmod against the strict YYYY-MM-DD pattern and mark bad formats.

- strptime parses a string against a format
- A wrong format raises ValueError
- try/except converts errors into verdicts

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 73: flag urls missing a lastmod. Project 75 changes the surface problem while preserving the underlying method.

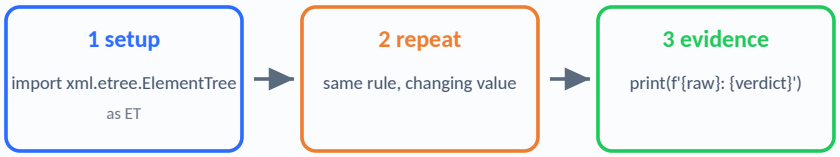
progress check

Reading target: validate lastmod as an iso date is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{raw}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for u in root.findall('url'):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{raw}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for u in root.findall('url'):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Read the try path as the hoped-for route and the catch/except/finally path as the boundary plan.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate lastmod as an ISO Date: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, datetime. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: for u in root.findall('url'): | Evidence: print(f'{raw}: {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain validate lastmod as an iso date in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 73.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 75 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{raw}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate lastmod as an iso date and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`datetime.strptime` with the `%Y-%m-%d` pattern raises `ValueError` on any string that does not match exactly, which is the validity test; note `strptime` parses a fixed string and never reads the current date.

```
import xml.etree.ElementTree as ET
from datetime import datetime
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><lastmod>2026-01-01</lastmod>
</url>'
       '<url><loc>https://example.com/b</loc><lastmod>2026/02/15</lastmod>
</url>'
```

```
        '<url><loc>https://example.com/c</loc><lastmod>15-03-2026</lastmod>
</url>'
        '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    raw = u.findtext('lastmod')
    try:
        datetime.strptime(raw, '%Y-%m-%d')
        verdict = 'valid'
    except ValueError:
        verdict = 'BAD FORMAT'
    print(f'{raw}: {verdict}')
```

```
output
2026-01-01: valid
2026/02/15: BAD FORMAT
15-03-2026: BAD FORMAT
```

How it works:

1. Three url nodes carry an ISO date, a slash date, and a reversed date.
2. For each, raw holds the lastmod text.
3. datetime.strptime(raw, '%Y-%m-%d') succeeds only for the ISO form.
4. The except branch sets verdict to BAD FORMAT, and each result prints.

Try it:

1. Accept a full timestamp form by trying a second strptime pattern on failure.
2. Tally how many dates pass versus fail and print a summary line.
3. Add a date with a two-digit year and observe the verdict.
4. Add `2026-1-1` and confirm `%Y-%m-%d` rejects the unpadded month.

⚠ Common Mistakes

- Using `datetime.now` to compare would be nondeterministic; `strptime` only parses the given string.
- A loose pattern like `%Y-%m-%d` still accepts a date plus a trailing time unless you strip it first.
- Catching a bare `except` would also swallow a `TypeError` when `raw` is `None`.

Skills gained strptime | try/except | date validation

→ Where You'll Use This: Date-format validation prevents silent rejection of lastmod values by search engines, a subtle failure that otherwise hides until indexing signals weaken.

Mental Model

strptime is a passport officer with one stamp shape: present a date in any other layout and it gets turned away at the desk.

Project 75 Check priority Is Within 0.0 to 1.0

The Idea Behind It

The priority tag must be a decimal between 0.0 and 1.0, and values outside that band are quietly ignored by crawlers. This code reads each priority, converts it with float, and applies a chained comparison to decide ok or OUT OF RANGE. Casting to float lets the numeric bounds check work directly on what XML stores only as text.

What You'll Build & Learn

Confirm each URL's priority falls within the legal 0.0 to 1.0 range.

- float casts text into a comparable number
- Chained comparison checks a numeric band
- Out-of-range priorities are ignored by crawlers

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 74: validate lastmod as an iso date. Project 76 changes the surface problem while preserving the underlying method.

progress check

Reading target: check priority is within 0.0 to 1.0 is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

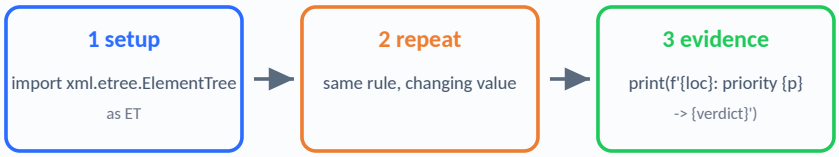
- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: priority {p} -> {verdict}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`verdict = 'ok' if 0.0 <= p <= 1.0 else 'OUT OF RANGE'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{loc}: priority {p} -> {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'ok' if 0.0 <= p <= 1.0 else 'OUT OF RANGE'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check priority Is Within 0.0 to 1.0: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: verdict = 'ok' if 0.0 <= p <= 1.0 else 'OUT OF RANGE' | Evidence: print(f'{loc}: priority {p} -> {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain check priority is within 0.0 to 1.0 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 74.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 76 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: priority {p} -> {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check priority is within 0.0 to 1.0 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Converting the text to `float` lets a single chained comparison `0.0 <= p <= 1.0` test the allowed band in one expression.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><priority>0.8</priority></url>'
       '<url><loc>https://example.com/b</loc><priority>1.5</priority></url>'
       '<url><loc>https://example.com/c</loc><priority>0.0</priority></url>'
       '</urlset>')
root = ET.fromstring(xml)
```

```
for u in root.findall('url'):
    loc = u.findtext('loc')
    p = float(u.findtext('priority'))
    verdict = 'ok' if 0.0 <= p <= 1.0 else 'OUT OF RANGE'
    print(f'{loc}: priority {p} -> {verdict}')
```

```
output
https://example.com/a: priority 0.8 -> ok
https://example.com/b: priority 1.5 -> OUT OF RANGE
https://example.com/c: priority 0.0 -> ok
```

How it works:

1. Three url nodes declare priorities 0.8, 1.5, and 0.0.
2. p = float(u.findtext('priority')) converts the text to a number.
3. The expression 0.0 <= p <= 1.0 decides ok versus OUT OF RANGE.
4. Each loc, its priority, and the verdict print together.

Try it:

1. Handle a missing priority by defaulting to 0.5 before the check.
2. Catch a non-numeric priority with try/except and label it MALFORMED.
3. Count how many URLs fall outside the legal band.
4. Add a `<priority>abc</priority>` and wrap the `float` in a try/except.

⚠ Common Mistakes

- `float('')` or `float(None)` raises, so a missing `<priority>` would crash without a guard.
- Comparing the raw string with `<=` would do lexicographic, not numeric, comparison.
- Priority is relative within your own site, not a ranking promise; out-of-range just gets ignored.

Skills gained float casting | chained comparison | range check

→ Where You'll Use This: Range-checking priority keeps a sitemap free of values crawlers discard, so the priority signals you do send actually reach the indexer intact.

Mental Model

Priority is a volume knob that only turns from 0 to 10: crank it past the stop and the amplifier simply ignores the impossible setting.

Project 76 Validate changefreq Against the Enum

The Idea Behind It

The changefreq tag accepts only seven keywords from always to never, so anything else is invalid. This project stores the legal words in a set, reads each url's changefreq, and tests membership with the in operator to find typos like often. Using a set makes each lookup constant time and expresses the closed enumeration directly in code.

What You'll Build & Learn

Validate every changefreq value against the seven-keyword allowed set.

- A set encodes a closed enumeration
- The in operator tests set membership
- changefreq has exactly seven legal values

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 75: check priority is within 0.0 to 1.0. Project 77 changes the surface problem while preserving the underlying method.

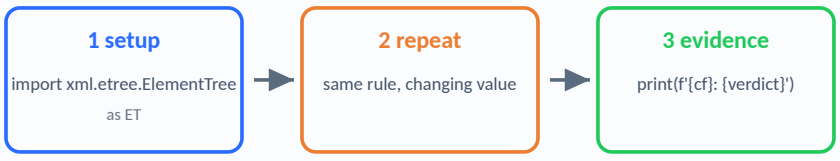
progress check

Reading target: validate changefreq against the enum is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{cf}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`verdict = 'ok' if cf in allowed else 'INVALID'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{cf}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'ok' if cf in allowed else 'INVALID'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate changefreq Against the Enum: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, allowed. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: verdict = 'ok' if cf in allowed else 'INVALID' | Evidence: print(f'{cf}: {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain validate changefreq against the enum in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 75.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 77 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{cf}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate changefreq against the enum and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A `set` of the seven legal keywords turns validation into a constant-time `in` membership test, rejecting any value that is not a recognised frequency.

```
import xml.etree.ElementTree as ET
allowed = {'always', 'hourly', 'daily', 'weekly', 'monthly', 'yearly', 'never'}
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><changefreq>daily</changefreq>
</url>'
       '<url><loc>https://example.com/b</loc><changefreq>often</changefreq>
</url>'
       '<url><loc>https://example.com/c</loc><changefreq>weekly</changefreq>
</url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    cf = u.findtext('changefreq')
    verdict = 'ok' if cf in allowed else 'INVALID'
    print(f'{cf}: {verdict}')
```

```
output
```

```
daily: ok
often: INVALID
weekly: ok
```

How it works:

1. allowed holds the seven valid keywords as a set.
2. Three url nodes declare daily, often, and weekly.
3. cf in allowed returns True only for real keywords.
4. Each value prints with ok or INVALID based on membership.

Try it:

1. Make the comparison case-insensitive by lowercasing cf first.
2. Collect the invalid values into a list and print them once.
3. Add a url with an empty changefreq and inspect the verdict.
4. Add `.lower()` so `WEEKLY` validates.

⚠ Common Mistakes

- Case matters: `Daily` would fail unless you lowercase first.
- A missing `<changefreq>` yields `None`, which is correctly not in the set but may deserve a separate message.
- Treating `changefreq` as a hard directive; crawlers treat it as a hint only.

Skills gained set membership | enum validation | findtext

→ Where You'll Use This: Validating changefreq against its enum catches editorial typos before they reach search engines, where unknown keywords are dropped and recrawl hints are lost.

⚡ Mental Model

The allowed set is a multiple-choice answer sheet: only the seven printed options score, and writing in your own answer earns nothing.

Project 77 Detect Duplicate loc Entries

💡 The Idea Behind It

A URL should appear once per sitemap, since duplicate loc values waste crawl budget and muddle reporting. This code feeds every loc into a Counter, then keeps the addresses whose tally exceeds one and sorts them. Counter does the frequency bookkeeping in a single pass, turning duplicate detection into a simple comprehension over its items.

What You'll Build & Learn

Detect and list the loc values that appear more than once in a urlset.

- Counter tallies occurrences in one pass
- Filter items by count greater than one
- Duplicate URLs waste crawl budget

Skill Roadmap

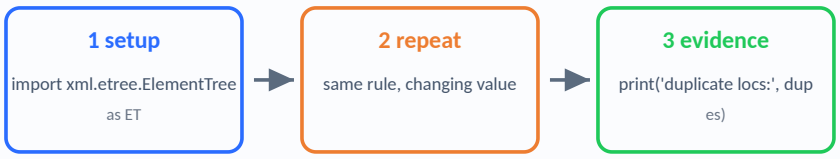
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 76: validate changefreq against the enum. Project 78 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect duplicate loc entries is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print('duplicate locs:', dupes)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build

Move	Code to inspect	Why it matters
		and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`dupes = sorted(loc for loc, n in counts.items() if n > 1)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('duplicate locs:', dupes)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dupes = sorted(loc for loc, n in counts.items() if n > 1)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Duplicate loc Entries: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, collections. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: dupes = sorted(loc for loc, n in counts.items() if n > 1) | Evidence: print('duplicate locs:', dupes)

Reader to Writer Bridge

1. Read: explain detect duplicate loc entries in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 76.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 78 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('duplicate locs:', dupes)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect duplicate loc entries and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `Counter` tallies how often each loc appears; filtering to counts above one and sorting yields the duplicate URLs in deterministic order.

```
import xml.etree.ElementTree as ET
from collections import Counter
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/c</loc></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
counts = Counter(u.findtext('loc') for u in root.findall('url'))
dupes = sorted(loc for loc, n in counts.items() if n > 1)
print('duplicate locs:', dupes)
```

```
output
duplicate locs: ['https://example.com/a', 'https://example.com/b']
```

How it works:

1. Five url nodes include /a and /b twice each.
2. Counter(...) tallies each loc's frequency from a generator.
3. The comprehension keeps locs where n > 1 and sorted orders them.
4. print shows duplicate locs as a sorted list.

Try it:

1. Print each duplicate alongside the number of times it appears.
2. Build a deduplicated loc list preserving first-seen order.
3. Report the percentage of entries that are duplicates.
4. Lowercase each loc before counting to catch case-variant duplicates.

⚠ Common Mistakes

- Building a `set` first would erase the duplicates before you could count them.
- Not sorting the result makes the printed order vary between runs.
- Case or trailing-slash variants are treated as distinct here, so true near-duplicates can slip through.

Skills gained `Counter` `generator expression` `deduplication`

→ Where You'll Use This: Spotting duplicate locs trims wasted recrawls and cleans up coverage reporting, a routine fix when sitemaps are stitched together from several sources.

⚡ Mental Model

Counter is a turnstile tally: it clicks once per visitor, and at day's end you read off who came back through more than once.

Project 78 Flag the 50,000-URL Limit

💡 The Idea Behind It

A single sitemap may hold at most fifty thousand URLs, and this project demonstrates the cap by generating an oversized list with a comprehension. It measures the list length, compares it to the LIMIT constant, and reports SPLIT NEEDED plus the exact overflow. No XML parsing is involved here; the lesson is purely the count-versus-cap arithmetic that decides when a file must be divided.

What You'll Build & Learn

Generate an over-limit URL list and report that it must be split, with the overflow.

- A range comprehension builds many URLs fast
- len compared to a cap gates the verdict
- Subtraction reports how far over the limit

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 77: detect duplicate loc entries. Project 79 changes the surface problem while preserving the underlying method.

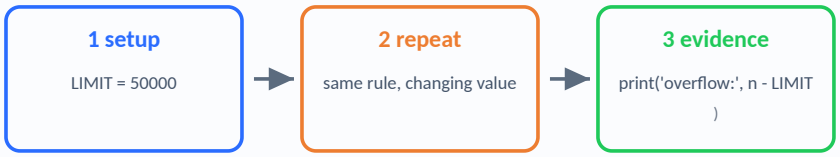
progress check

Reading target: flag the 50,000-url limit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `LIMIT = 50000` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print('overflow:', n - LIMIT)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`LIMIT = 50000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`verdict = 'SPLIT NEEDED' if n > LIMIT else 'ok'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('overflow:', n - LIMIT)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'SPLIT NEEDED' if n > LIMIT else 'ok'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag the 50,000-URL Limit: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: urls, https, example, com, range. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: LIMIT = 50000 | Core: verdict = 'SPLIT NEEDED' if n > LIMIT else 'ok' | Evidence: print('overflow:', n - LIMIT)

Reader to Writer Bridge

1. Read: explain flag the 50,000-url limit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 77.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 79 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('overflow:', n - LIMIT)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `LIMIT = 50000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag the 50,000-url limit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Comparing the generated list length to the 50,000 cap detects when a urlset exceeds the protocol limit and must be partitioned across several sitemaps under an index.

```
LIMIT = 50000
urls = [f'https://example.com/p/{i}' for i in range(50002)]
n = len(urls)
verdict = 'SPLIT NEEDED' if n > LIMIT else 'ok'
print(f'{n} URLs (limit {LIMIT}) -> {verdict}')
print('overflow:', n - LIMIT)
```

```
output
50002 URLs (limit 50000) -> SPLIT NEEDED
overflow: 2
```

How it works:

1. A comprehension builds 50002 fake URLs with range.
2. n = len(urls) measures the list size.
3. n > LIMIT decides SPLIT NEEDED versus ok.
4. The code prints the count, the verdict, and n minus LIMIT as overflow.

Try it:

1. Compute how many capped files the overflow requires using ceiling division.
2. Lower LIMIT to a tiny value and confirm the split triggers.
3. Print the index of the first URL that exceeds the cap.
4. Lower `LIMIT` to 50000 and a list of 50000 to confirm it just passes.

⚠ Common Mistakes

- The limit is 50,000 URLs OR 50 MB uncompressed, whichever comes first; this checks only the count.
- Off-by-one: `range(50002)` produces exactly 50002 items, so `> LIMIT` is the right comparator, not `>=`.
- Counting `<loc>` plus child `<image>` locs against the URL limit double-counts; only `<url>` entries count.

Skills gained `range` `list comprehension` `limit check`

→ **Where You'll Use This:** The fifty-thousand cap forces large sites to shard their sitemaps and publish an index, so this overflow check decides exactly how many files you need.

⚡ Mental Model

Fifty thousand is an elevator's posted capacity: one rider over the placard and someone has to wait for the next car.

Project 79 Parse a Sitemap Index of Children

💡 The Idea Behind It

Large sites use a sitemapindex that points to many child sitemaps, each with its own loc and lastmod. This project parses such an index, iterates its sitemap children, and prints each child's location alongside its modification date, then counts them. Reading the index is how a crawler fans out from one entry point to every shard of a sprawling site.

What You'll Build & Learn

Parse a sitemapindex and print each child sitemap's loc and lastmod, plus the count.

- A sitemapindex lists child sitemaps
- Each sitemap child has loc and lastmod
- findall iterates the sitemap nodes

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 78: flag the 50,000-url limit. Project 80 changes the surface problem while preserving the underlying method.

progress check

Reading target: parse a sitemap index of children is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.

- Evidence: `print('child sitemaps:', len(root.findall('sitemap')))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for sm in root.findall('sitemap'):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('child sitemaps:', len(root.findall('sitemap')))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for sm in root.findall('sitemap'):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Parse a Sitemap Index of Children: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, sitemapindex. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: for sm in root.findall('sitemap'): | Evidence: print('child sitemaps:', len(root.findall('sitemap')))

🛠️ Reader to Writer Bridge

1. Read: explain parse a sitemap index of children in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 78.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 80 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('child sitemaps:', len(root.findall('sitemap')))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of parse a sitemap index of children and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A sitemap index uses `<sitemap>` children (not `<url>`), each carrying a `<loc>` pointing at a child sitemap file, so the parsing mirrors a urlset with different tag names.

```
import xml.etree.ElementTree as ET
xml = ('<sitemapindex>'
       '<sitemap><loc>https://example.com/sitemap-posts.xml</loc><lastmod>2026-
01-10</lastmod></sitemap>'
```

```
        '<sitemap><loc>https://example.com/sitemap-pages.xml</loc><lastmod>2026-
02-01</lastmod></sitemap>'
        '</sitemapindex>')
root = ET.fromstring(xml)
for sm in root.findall('sitemap'):
    print(sm.findtext('loc'), '->', sm.findtext('lastmod'))
print('child sitemaps:', len(root.findall('sitemap')))
```

```
output
https://example.com/sitemap-posts.xml -> 2026-01-10
https://example.com/sitemap-pages.xml -> 2026-02-01
child sitemaps: 2
```

How it works:

1. ET.fromstring parses the sitemapindex into a root element.
2. root.findall('sitemap') yields two sitemap children.
3. For each, findtext reads loc and lastmod, printed with an arrow between.
4. A final print reports child sitemaps: 2.

Try it:

1. Validate each child loc starts with https before printing it.
2. Sort the children by lastmod to surface the freshest shard.
3. Flag any child sitemap whose lastmod child is absent.
4. Add a third child sitemap and watch the count rise.

⚠ Common Mistakes

- Looking for `<url>` in an index returns nothing; the entries are `<sitemap>`.
- A `<loc>` in an index must point to another sitemap file, never to a page.
- Mixing `<url>` and `<sitemap>` entries in one file is invalid; a file is either an index or a urlset.

Skills gained `sitemapindex` `findall` `findtext`

→ Where You'll Use This: Reading the index is how crawlers and audit tools discover every shard of a big site, so this fan-out step underpins full-coverage analysis.

Mental Model

A sitemapindex is a table of contents: it does not hold the chapters, just the page numbers that tell you where each one begins.

Project 80 Build a Urlset From a URL List

The Idea Behind It

Auditing is only half the job; the other half is generating valid XML. This project takes a plain Python list of URLs, opens with a urlset tag, appends one url and loc block per address inside a loop, then closes the tag and joins everything into one string. Building the parts in a list and joining at the end is the efficient way to assemble markup.

What You'll Build & Learn

Wrap a list of URLs into a well-formed urlset XML string.

- Build markup as a list of string parts
- An f-string templates each url block
- join concatenates the pieces efficiently

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 79: parse a sitemap index of children. Project 81 changes the surface problem while preserving the underlying method.

progress check

Reading target: build a urlset from a url list is the surface task; the transferable Python move is to mark setup, rule, and output before running.

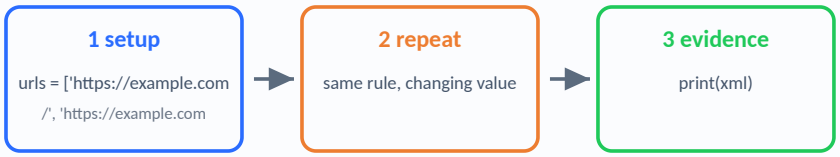
setup -> rule -> result

Logic Lens

- Setup: `urls = ['https://example.com/', 'https://example.com/blog', 'https://example.com/shop']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(xml)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = ['https://example.com/', 'https://example.com/blog', 'https://example.com/shop']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for u in urls:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(xml)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for u in urls:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a Urlset From a URL List: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: urls, https, example, com, blog. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: urls = ['https://example.com/', 'https://example.com/blog', 'https://ex... | Core: for u in urls: | Evidence: print(xml)

🛠️ Reader to Writer Bridge

1. Read: explain build a urlset from a url list in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 79.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 81 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(xml)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `urls = ['https://example.com/', 'https://example.com/blog', 'https://example.com/shop']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a urlset from a url list and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Accumulating fragments in a list and joining once builds the urlset efficiently; a real sitemap would add the `xmlns` attribute on `<urlset>` but the structure is otherwise identical.

```
urls = ['https://example.com/', 'https://example.com/blog',
'https://example.com/shop']
parts = ['<urlset>']
for u in urls:
    parts.append(f'<url><loc>{u}</loc></url>')
parts.append('</urlset>')
```

```
xml = ''.join(parts)
print(xml)
```

```
output
<urlset><url><loc>https://example.com/</loc></url><url>
<loc>https://example.com/blog</loc></url><url><loc>https://example.com/shop</loc>
</url></urlset>
```

How it works:

1. parts starts as a list holding the opening urlset tag.
2. The loop appends a url and loc block for each address via an f-string.
3. parts.append closes the urlset tag.
4. ''.join(parts) fuses the list into one XML string that prints.

Try it:

1. Add a lastmod child to each url block from a parallel list.
2. Escape ampersands in each URL before embedding it.
3. Pretty-print the output with newlines between url blocks.
4. Add a `<lastmod>` child inside each `<url>` block.

⚠ Common Mistakes

- String-concatenating in a loop with `+=` is quadratic; building a list and joining is linear.
- URLs containing `&` must be escaped to `&` first or the XML is malformed.
- Omitting the `xmlns` makes this fixture parse here but invalid as a live sitemap.

Skills gained string building | join | XML generation

→ Where You'll Use This: Generating a urlset is the core of any sitemap publisher, the step that turns a crawl or CMS export into the file you submit to search engines.

Mental Model

Building parts then joining is like prepping ingredients in bowls before stirring once: assembling piecemeal beats restitching the whole string each loop.

Project 81 Build a Sitemap Index File

The Idea Behind It

When you split URLs across many sitemaps, you publish one index listing them all. This project mirrors the urlset builder but emits a sitemapindex: it opens the index tag, appends a sitemap and loc block per child URL,

closes the tag, and joins the parts. It is the final assembly step that ties a fleet of child files together for crawlers.

What You'll Build & Learn

Generate a sitemapindex XML string from a list of child sitemap URLs.

- An index wraps sitemap not url elements
- Loop and append builds each child block
- join produces the final index string

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 80: build a urlset from a url list. Project 82 changes the surface problem while preserving the underlying method.

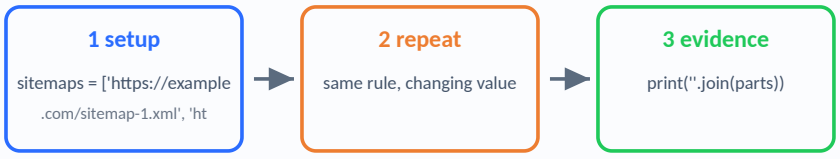
progress check

Reading target: build a sitemap index file is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `sitemaps = ['https://example.com/sitemap-1.xml', 'https://example.com/sitemap-2.xml']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(''.join(parts))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`sitemaps = ['https://example.com/sitemap-1.xml', 'https://example.com/sitemap-2.xml']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for s in sitemaps:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(''.join(parts))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for s in sitemaps:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a Sitemap Index File: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: sitemaps, https, example, com, sitemap. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: sitemaps = ['https://example.com/sitemap-1.xml', 'https://example.com/s... | Core: for s in sitemaps: | Evidence: print(''.join(parts))

🛠️ Reader to Writer Bridge

1. Read: explain build a sitemap index file in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 80.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 82 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(''.join(parts))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `sitemaps = ['https://example.com/sitemap-1.xml', 'https://example.com/sitemap-2.xml']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a sitemap index file and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The index mirrors a urlset but uses `<sitemap>`/`<loc>` pairs pointing at child files; this is what you submit to search engines once a site outgrows one sitemap.

```
sitemaps = ['https://example.com/sitemap-1.xml', 'https://example.com/sitemap-2.xml']
parts = ['<sitemapindex>']
for s in sitemaps:
    parts.append(f'<sitemap><loc>{s}</loc></sitemap>')
parts.append('</sitemapindex>')
print(''.join(parts))
```

```
output
<sitemapindex><sitemap><loc>https://example.com/sitemap-1.xml</loc></sitemap>
<sitemap><loc>https://example.com/sitemap-2.xml</loc></sitemap></sitemapindex>
```

How it works:

1. parts begins with the opening sitemapindex tag.
2. The loop appends a sitemap and loc block for each child URL.
3. parts.append closes the sitemapindex tag.
4. ''.join(parts) prints the complete index XML.

Try it:

1. Add a lastmod child to each sitemap block from a parallel list.
2. Validate every child URL is absolute https before appending it.
3. Reuse a helper that takes the tag names so one function builds both file types.
4. Add `<lastmod>` to each `<sitemap>` entry.

⚠ Common Mistakes

- Each child loc must be a full absolute URL, not a relative path.
- An index may not nest another index; it only lists leaf sitemaps.
- Forgetting the closing `</sitemapindex>` produces XML that fails to parse.

Skills gained sitemapindex | string building | join

→ Where You'll Use This: Generating the index is the capstone of sharding a large sitemap, producing the single file you actually submit so crawlers can reach every child.

⚡ Mental Model

The index file is a building directory in the lobby: it lists which floor each department occupies so visitors know where to head next.

Project 82 Require Absolute HTTPS Locations

💡 The Idea Behind It

Sitemap URLs must be fully qualified and absolute, never relative paths. This project parses a urlset and tests whether each loc begins with the https scheme, labelling anything that does not as NOT ABSOLUTE HTTPS. The startswith check rejects bare paths like /relative and schemeless hosts like example.com, both of which crawlers cannot resolve from inside a sitemap.

What You'll Build & Learn

Verify each loc is an absolute https URL and flag relative or schemeless ones.

- Sitemap locs must be absolute, not relative
- startswith tests a required prefix

- Schemeless hosts are invalid entries

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 81: build a sitemap index file. Project 83 changes the surface problem while preserving the underlying method.

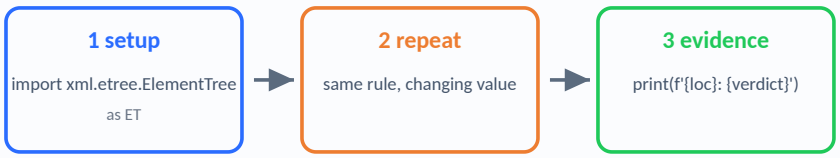
progress check

Reading target: require absolute https locations is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`verdict = 'ok' if loc.startswith('https://') else 'NOT ABSOLUTE HTTPS'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'{loc}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'ok' if loc.startswith('https://') else 'NOT ABSOLUTE HTTPS'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Require Absolute HTTPS Locations: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: verdict = 'ok' if loc.startswith('https://') else 'NOT ABSOLUTE HTTPS' | Evidence: print(f'{loc}: {verdict}')

Reader to Writer Bridge

1. Read: explain require absolute https locations in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 81.
   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 83 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of require absolute https locations and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A simple `startswith('https://')` prefix test enforces that every loc is an absolute, secure URL, the form the sitemap protocol requires.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>/relative/path</loc></url>'
       '<url><loc>example.com/no-scheme</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    verdict = 'ok' if loc.startswith('https://') else 'NOT ABSOLUTE HTTPS'
    print(f'{loc}: {verdict}')
```

```
output
https://example.com/a: ok
/relative/path: NOT ABSOLUTE HTTPS
example.com/no-scheme: NOT ABSOLUTE HTTPS
```

How it works:

1. The urlset holds an absolute URL, a relative path, and a schemeless host.
2. For each url, loc is read with findtext.
3. loc.startswith('https://') decides ok versus NOT ABSOLUTE HTTPS.
4. Each loc prints with its verdict.

Try it:

1. Also accept http during a migration and note which scheme each uses.
2. Use urllib.parse to confirm both a scheme and a netloc are present.

3. Collect only the invalid locs for a remediation list.
4. Accept either https or an uppercase scheme by lowercasing the prefix check.

⚠ Common Mistakes

- A protocol-relative `//example.com/x` is not absolute and rightly fails this test.
- Allowing any scheme by checking only for `://` would pass ftp and mailto links.
- Whitespace around the loc would defeat `startswith`; strip first if your source is messy.

Skills gained `startswith` `absolute URLs` `validation`

→ **Where You'll Use This:** Rejecting relative or schemeless locs prevents crawlers from discarding entries they cannot resolve, a frequent defect when sitemaps are templated by hand.

⚡ Mental Model

An absolute URL is a full mailing address with city and country; a relative path is just an apartment number that means nothing without the building.

Project 83 Detect a url Entry With No loc

💡 The Idea Behind It

Every url element must contain exactly one loc; a block without it is invalid and unparseable for crawlers. This project enumerates the url nodes, reads loc with findtext, and tests for None to spot the empty entry. Using enumerate gives a positional index so the report can pinpoint exactly which url block is malformed.

What You'll Build & Learn

Find any url entry that lacks a loc child and report its position.

- findtext returns None for a missing loc
- enumerate supplies a positional index
- Every url needs exactly one loc

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 82: require absolute https locations. Project 84 changes the surface problem while preserving the underlying method.

progress check

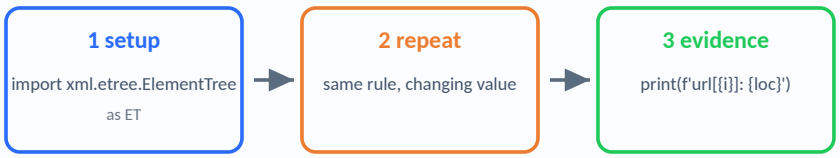
Reading target: detect a url entry with no loc is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'url[{i}]: {loc}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`if loc is None:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'url[{i}]: {loc}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if loc is None:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a url Entry With No loc: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: if loc is None: | Evidence: print(f'url[{i}]: {loc}')

🛠️ Reader to Writer Bridge

1. Read: explain detect a url entry with no loc in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 82.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 84 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'url[{i}]: {loc}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of detect a url entry with no loc and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`findtext` returns `None` when `<loc>` is absent, so an `is None` test isolates url blocks that violate the one-loc-per-url rule.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><lastmod>2026-01-01</lastmod></url>'
       '<url><loc>https://example.com/c</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
for i, u in enumerate(root.findall('url')):
    loc = u.findtext('loc')
    if loc is None:
        print(f'url[{i}]: MISSING loc (invalid entry)')
    else:
        print(f'url[{i}]: {loc}')
```

```
output
url[0]: https://example.com/a
url[1]: MISSING loc (invalid entry)
url[2]: https://example.com/c
```

How it works:

1. Three url nodes appear; the middle one has lastmod but no loc.
2. enumerate pairs each url with its index i.
3. loc is None for the second entry, triggering the MISSING loc branch.
4. Valid entries print their index and URL instead.

Try it:

1. Count the invalid entries and print a total at the end.
2. Also flag url blocks that contain two loc children.
3. Skip the invalid entries while collecting the valid locs into a list.
4. Also flag entries that contain more than one `<loc>` via `len(u.findall('loc'))`.

⚠ Common Mistakes

- Checking `if not loc` would also flag a present-but-empty `<loc></loc>`, which may be a different error.
- `enumerate` indexes are positional, not the URL itself, so report the loc when you can.

- A `<url>` with two `<loc>` tags is also invalid but this check only catches zero.

Skills gained `enumerate` `None check` `findtext`

→ **Where You'll Use This:** Catching loc-less url blocks keeps a sitemap parseable, since search engines may reject the whole file when even one entry breaks the required structure.

Mental Model

A url without a loc is an envelope with no address line: the post office cannot route it no matter what else is written inside.

Project 84 Escape Ampersands in a URL

The Idea Behind It

A raw ampersand in a URL breaks XML parsing because the parser reads it as the start of an entity. This project uses xml.sax.saxutils.escape to convert each ampersand in a query string into the entity & before the URL is embedded. Escaping reserved characters is what keeps a generated sitemap well-formed even when URLs carry multiple parameters.

What You'll Build & Learn

Escape ampersands in query-string URLs so the resulting XML stays well-formed.

- Raw ampersands break XML parsing
- saxutils.escape encodes reserved characters
- & is the safe encoding of &

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 83: detect a url entry with no loc. Project 85 changes the surface problem while preserving the underlying method.

progress check

Reading target: escape ampersands in a url is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

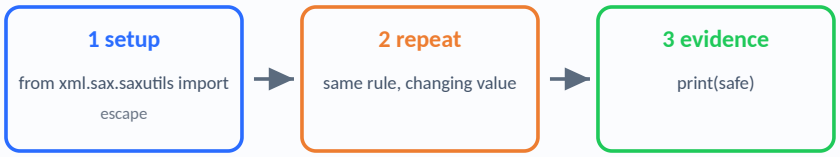
Logic Lens

- Setup: `from xml.sax.saxutils import escape` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(safe)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from xml.sax.saxutils import escape`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for u in urls:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(safe)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for u in urls:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Escape Ampersands in a URL: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, sax, saxutils, escape, urls. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from xml.sax.saxutils import escape | Core: for u in urls: | Evidence: print(safe)

🛠️ Reader to Writer Bridge

1. Read: explain escape ampersands in a url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 83.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 85 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(safe)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from xml.sax.saxutils import escape` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of escape ampersands in a url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`xml.sax.saxutils.escape` converts the XML-special characters `&`, `<` and `>` into entities, so a query string with `&` can be safely placed inside a `<loc>`.

```
from xml.sax.saxutils import escape
urls = [
    'https://example.com/search?q=shoes&sort=price',
    'https://example.com/page?a=1&b=2&c=3',
]
for u in urls:
    safe = escape(u)
    print(safe)
```

```
output
https://example.com/search?q=shoes&amp;sort=price
https://example.com/page?a=1&amp;b=2&amp;c=3
```

How it works:

1. Two URLs carry query strings joined by raw ampersands.
2. escape(u) replaces each & with the entity &.
3. The loop prints each escaped URL.
4. Output shows sort=price and the chained parameters now safely encoded.

Try it:

1. Add a URL containing a literal less-than sign and confirm it is escaped too.
2. Round-trip by unescaping the result and checking it matches the input.
3. Embed each escaped URL inside a full loc element.
4. Add a URL containing `<` or `>` and watch them become `<`/`>`.

⚠ Common Mistakes

- Replacing `&` by hand can double-escape an already-encoded `&` into `&`.
- `escape` does not touch quotes by default; pass a mapping if attribute values need it.
- Forgetting to escape before building the XML yields a `ParseError` at read time.

Skills gained saxutils escape | XML entities | query strings

→ Where You'll Use This: Escaping reserved characters is mandatory when emitting sitemaps for sites with parameterised URLs, since one unescaped ampersand can invalidate the entire file.

Mental Model

An unescaped ampersand is a live wire in the XML circuit: escape turns it into insulated & so the parser does not short out.

Project 85 Flag URLs Longer Than 2048 Chars

The Idea Behind It

Sitemap locs have a practical ceiling of 2048 characters, and search engines may reject anything longer. This project builds an oversized URL by repeating a character 2050 times, embeds it in a urlset, then measures each loc's length and flags any over the limit. The string-multiplication trick fabricates a runaway URL so the length guard has something to catch.

What You'll Build & Learn

Measure each loc's character length and flag URLs longer than 2048.

- String multiplication fabricates a long URL
- len measures a loc's character count
- URLs over 2048 chars may be rejected

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 84: escape ampersands in a url. Project 86 changes the surface problem while preserving the underlying method.

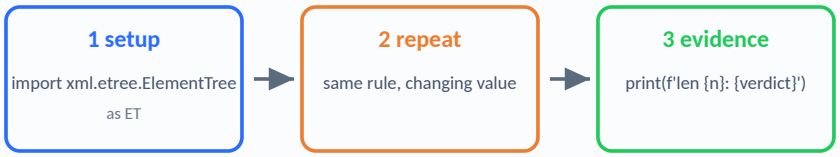
progress check

Reading target: flag urls longer than 2048 chars is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'len {n}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`verdict = 'TOO LONG' if n > 2048 else 'ok'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'len {n}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'TOO LONG' if n > 2048 else 'ok'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag URLs Longer Than 2048 Chars: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, long_path. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: verdict = 'TOO LONG' if n > 2048 else 'ok' | Evidence: print(f'len {n}: {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain flag urls longer than 2048 chars in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 84.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 86 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'len {n}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag urls longer than 2048 chars and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`len(loc)` measures the character count of each URL, and comparing against 2048 flags entries that exceed the conventional URL-length limit.

```
import xml.etree.ElementTree as ET
long_path = 'x' * 2050
xml = ('<urlset>'
       f'<url><loc>https://example.com/{long_path}</loc></url>'
       '<url><loc>https://example.com/short</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
```

```
for u in root.findall('url'):
    loc = u.findtext('loc')
    n = len(loc)
    verdict = 'TOO LONG' if n > 2048 else 'ok'
    print(f'len {n}: {verdict}')
```

```
output
len 2070: TOO LONG
len 25: ok
```

How it works:

1. long_path repeats x 2050 times to build an over-length segment.
2. That path is embedded in a urlset beside a short URL.
3. n = len(loc) measures each loc, and n > 2048 sets TOO LONG.
4. The lengths 2070 and 25 print with their verdicts.

Try it:

1. Print the full loc only when it is under the limit to keep output short.
2. Lower the ceiling to test the boundary at exactly 2048 characters.
3. Report the longest loc in the set and its length.
4. Lower the threshold to 2000 and watch the short URL still pass.

⚠ Common Mistakes

- The limit is on characters, but very long multibyte URLs may also blow the byte budget.
- Building the fixture path with `'x' * 2050` makes the total length 2070 once the prefix is added, not 2050.
- A few long URLs are usually a CMS bug (endless faceted params) rather than real content.

Skills gained len | string multiplication | length limit

→ Where You'll Use This: Length guards catch runaway URLs produced by deep faceted navigation or stacked parameters, which crawlers may silently drop from the index.

⚡ Mental Model

The 2048 ceiling is a doorframe height: most URLs walk through fine, but the occasional giant has to duck or gets turned away.

Project 86 Sort URLs by lastmod, Newest First

💡 The Idea Behind It

Editors want to see what changed most recently, so this project sorts URLs by their lastmod in descending order. It builds a list of loc and lastmod tuples from the urlset, then sorts using a key lambda on the date field with reverse set to True. Because ISO dates sort lexicographically the same way they sort chronologically, plain string comparison suffices.

What You'll Build & Learn

Sort the sitemap's URLs by lastmod, newest first, and print them.

- Build tuples of loc and lastmod
- A key lambda sorts on the date field
- ISO dates sort correctly as strings

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 85: flag urls longer than 2048 chars. Project 87 changes the surface problem while preserving the underlying method.

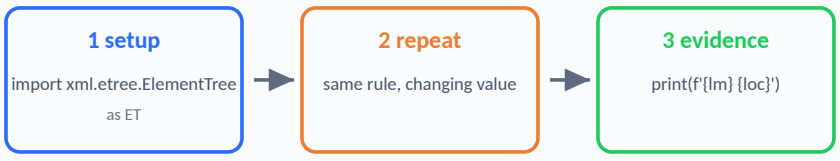
progress check

Reading target: sort urls by lastmod, newest first is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{lm} {loc}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`rows = [(u.findtext('loc'), u.findtext('lastmod')) for u in root.findall('url')]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{lm} {loc}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rows = [(u.findtext('loc'), u.findtext('lastmod')) for u in root.findall('url')]`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Sort URLs by lastmod, Newest First: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.

Layer	What to read
Evidence	Entry: import xml.etree.ElementTree as ET | Core: rows = [(u.findtext('loc'), u.findtext('lastmod')) for u in root.findal... | Evidence: print(f'{lm} {loc}')

🛠️ Reader to Writer Bridge

1. Read: explain sort urls by lastmod, newest first in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 85.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 87 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{lm} {loc}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sort urls by lastmod, newest first and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Because ISO `YYYY-MM-DD` dates sort identically as strings and as dates, a plain `reverse=True` string sort on the lastmod field orders pages newest-first without parsing.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><lastmod>2026-01-05</lastmod>
</url>'
       '<url><loc>https://example.com/b</loc><lastmod>2026-03-20</lastmod>
</url>'
       '<url><loc>https://example.com/c</loc><lastmod>2026-02-10</lastmod>
</url>'
       '</urlset>')
root = ET.fromstring(xml)
rows = [(u.findtext('loc'), u.findtext('lastmod')) for u in
root.findall('url')]
for loc, lm in sorted(rows, key=lambda r: r[1], reverse=True):
    print(f'{lm}  {loc}')
```

```
output
2026-03-20  https://example.com/b
2026-02-10  https://example.com/c
2026-01-05  https://example.com/a
```

How it works:

1. A comprehension builds rows of loc and lastmod tuples.
2. sorted with key=lambda r: r[1] orders by the date.
3. reverse=True puts the newest date first.
4. Each row prints its lastmod then its loc, newest at the top.

Try it:

1. Sort ascending to find the oldest pages first.
2. Handle URLs with no lastmod by sorting them to the bottom.
3. Limit the output to the five most recent URLs.
4. Add a URL with no lastmod and default it to an empty string before sorting.

⚠ Common Mistakes

- String sorting only works because the dates are zero-padded ISO; `2026-3-20` would sort wrong.
- A `None` lastmod would crash the comparison; filter or default missing dates first.
- Forgetting `reverse=True` lists the oldest pages first instead.

Skills gained sorted | key lambda | tuples

→ Where You'll Use This: Sorting by recency surfaces the freshest pages for editorial review and helps verify that recently updated content is actually reflected in the sitemap.

⚡ Mental Model

Sorting by lastmod is stacking the newspaper pile so today's edition sits on top and last month's settles to the bottom.

Project 87 Flag Stale lastmod Against an As-Of Date

💡 The Idea Behind It

Pages untouched for months may signal neglected content, so this project flags any whose lastmod predates a staleness window. It parses a fixed AS_OF date, then for each url computes the day gap to its lastmod by subtracting datetimes and reading the timedelta's days attribute. Comparing that age against STALE_DAYS turns a fuzzy notion of freshness into a measured threshold.

What You'll Build & Learn

Flag URLs whose lastmod is older than a staleness window relative to a fixed date.

- Subtracting datetimes yields a timedelta
- The days attribute gives the age gap
- A threshold turns age into a flag

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 86: sort urls by lastmod, newest first. Project 88 changes the surface problem while preserving the underlying method.

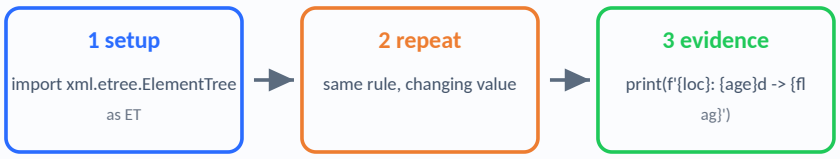
progress check

Reading target: flag stale lastmod against an as-of date is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {age}d -> {flag}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build

Move	Code to inspect	Why it matters
		and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`flag = 'STALE' if age > STALE_DAYS else 'fresh'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: {age}d -> {flag}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = 'STALE' if age > STALE_DAYS else 'fresh'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Stale lastmod Against an As-Of Date: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, datetime. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: flag = 'STALE' if age > STALE_DAYS else 'fresh' | Evidence: print(f'{loc}: {age}d -> {flag}')

Reader to Writer Bridge

1. Read: explain flag stale lastmod against an as-of date in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 86.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 88 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {age}d -> {flag}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag stale lastmod against an as-of date and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Parsing both the lastmod and a fixed `AS_OF` constant with `strptime` lets a `timedelta.days` subtraction measure each page's age deterministically, with no call to the real clock.

```
import xml.etree.ElementTree as ET
from datetime import datetime
AS_OF = '2026-06-01'
STALE_DAYS = 90
cutoff = datetime.strptime(AS_OF, '%Y-%m-%d')
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><lastmod>2026-05-20</lastmod>
</url>'
       '<url><loc>https://example.com/b</loc><lastmod>2026-01-01</lastmod>
</url>'
       '<url><loc>https://example.com/c</loc><lastmod>2025-09-01</lastmod>
</url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    lm = datetime.strptime(u.findtext('lastmod'), '%Y-%m-%d')
    age = (cutoff - lm).days
    flag = 'STALE' if age > STALE_DAYS else 'fresh'
    print(f'{loc}: {age}d -> {flag}')
```

```
output
https://example.com/a: 12d -> fresh
https://example.com/b: 151d -> STALE
https://example.com/c: 273d -> STALE
```

How it works:

1. cutoff is the AS_OF date parsed with strptime.
2. Each url's lastmod is parsed to a datetime named lm.
3. (cutoff - lm).days computes the age in days.
4. age greater than STALE_DAYS prints STALE, otherwise fresh.

Try it:

1. Make STALE_DAYS configurable and rerun with a 30-day window.
2. Sort the stale pages oldest first for a refresh queue.
3. Use today's date instead of a fixed AS_OF constant.
4. Raise `STALE_DAYS` to 200 and watch URL b flip to fresh.

⚠ Common Mistakes

- Using `datetime.now` here would make the output change daily; a fixed `AS_OF` keeps it reproducible.
- A future lastmod gives a negative age that is never stale, which may itself be a data error worth flagging.
- Comparing the date strings directly would tell you order but not the age in days.

Skills gained datetime arithmetic | timedelta | threshold check

→ Where You'll Use This: Staleness reports prioritise refresh work by surfacing pages that have drifted out of date, a common input to content-pruning and update campaigns.

⚡ Mental Model

STALE_DAYS is a milk expiry rule: anything past the window goes in the review bin, no matter how fine the carton looks from outside.

Project 88 Find Sitemap URLs Not in the Crawl

💡 The Idea Behind It

A URL listed in the sitemap but never reached by the crawler is suspicious, since it may be orphaned or blocked. This project builds a set from the sitemap's locs and another representing crawled pages, then subtracts the crawled set to expose the orphans. Set difference does in one operation what would otherwise be a nested membership loop.

What You'll Build & Learn

Find URLs declared in the sitemap that the crawler never visited.

- A set comprehension collects the sitemap locs
- Set subtraction finds the difference
- Sitemap-only URLs may be orphaned

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 87: flag stale lastmod against an as-of date. Project 89 changes the surface problem while preserving the underlying method.

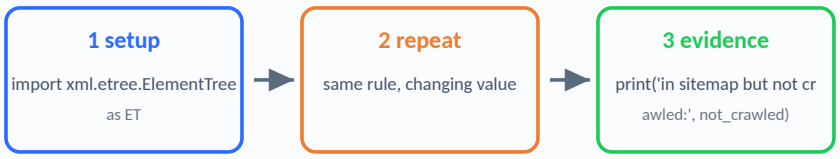
progress check

Reading target: find sitemap urls not in the crawl is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print('in sitemap but not crawled:', not_crawled)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build

Move	Code to inspect	Why it matters
		and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`sitemap = {u.findtext('loc') for u in root.findall('url')}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('in sitemap but not crawled:', not_crawled)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sitemap = {u.findtext('loc') for u in root.findall('url')}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Sitemap URLs Not in the Crawl: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: sitemap = {u.findtext('loc') for u in root.findall('url')} | Evidence: print('in sitemap but not crawled:', not_crawled)

Reader to Writer Bridge

1. Read: explain find sitemap urls not in the crawl in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 87.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 89 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('in sitemap but not crawled:', not_crawled)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find sitemap urls not in the crawl and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Subtracting the crawled set from the sitemap set yields URLs the sitemap promised but the crawler never fetched, a classic orphan or broken-link signal.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '<url><loc>https://example.com/orphan</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
sitemap = {u.findtext('loc') for u in root.findall('url')}
crawled = {'https://example.com/a', 'https://example.com/b',
'https://example.com/extra'}
not_crawled = sorted(sitemap - crawled)
print('in sitemap but not crawled:', not_crawled)
```

```
output
in sitemap but not crawled: ['https://example.com/orphan']
```

How it works:

1. A set comprehension gathers the three sitemap locs.
2. crawled is a literal set of pages the crawler reached.
3. sitemap - crawled keeps locs absent from the crawl.
4. sorted prints the single orphan, /orphan.

Try it:

1. Print the count of sitemap-only URLs alongside the list.
2. Read the crawled set from a file instead of a literal.
3. Show the intersection as the URLs found in both.
4. Print the crawled-but-not-in-sitemap set as well with the operands swapped.

⚠ Common Mistakes

- Reversing the operands answers the opposite question; order matters in set difference.
- URL normalization differences (trailing slash, case) cause false mismatches between the two sets.
- Not sorting the difference gives nondeterministic output order.

Skills gained set difference | set comprehension | crawl comparison

→ Where You'll Use This: Comparing sitemap declarations against an actual crawl exposes orphaned or blocked pages, one of the most actionable findings in a technical-SEO audit.

⚡ Mental Model

Set subtraction is a guest list against the door scanner: names on the list that never beeped in are the no-shows worth chasing.

Project 89 Find Crawled URLs Missing From the Sitemap

💡 The Idea Behind It

The reverse gap matters too: pages the crawler found but the sitemap omits should probably be added. This project builds a set of sitemap locs, then subtracts it from the crawled set so the leftover URLs are exactly those missing from the sitemap. It is the mirror image of the orphan check, reusing set difference in the opposite direction.

What You'll Build & Learn

List crawled URLs that the sitemap fails to declare.

- Set difference runs in either direction
- crawled minus sitemap finds omissions
- Missing URLs are candidates to add

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 88: find sitemap urls not in the crawl. Project 90 changes the surface problem while preserving the underlying method.

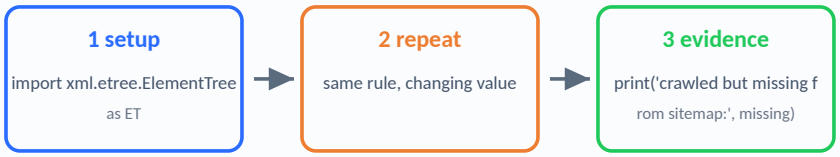
progress check

Reading target: find crawled urls missing from the sitemap is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print('crawled but missing from sitemap:', missing)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`sitemap = {u.findtext('loc') for u in root.findall('url')}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('crawled but missing from sitemap:', missing)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sitemap = {u.findtext('loc') for u in root.findall('url')}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Crawled URLs Missing From the Sitemap: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: sitemap = {u.findtext('loc') for u in root.findall('url')} | Evidence: print('crawled but missing from sitemap:', missing)

🛠️ Reader to Writer Bridge

1. Read: explain find crawled urls missing from the sitemap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 88.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 90 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('crawled but missing from sitemap:', missing)`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find crawled urls missing from the sitemap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

`crawled - sitemap` returns URLs discovered by following links that the sitemap never declared, the candidates for inclusion to improve discovery.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
sitemap = {u.findtext('loc') for u in root.findall('url')}
crawled = {'https://example.com/a', 'https://example.com/b',
'https://example.com/c', 'https://example.com/d'}
missing = sorted(crawled - sitemap)
print('crawled but missing from sitemap:', missing)
```

```
output
crawled but missing from sitemap: ['https://example.com/c', 'https://example.com/d']
```

How it works:

1. A set comprehension collects the two sitemap locs.
2. crawled is a literal set of four reached pages.
3. crawled - sitemap keeps the pages the sitemap omits.
4. sorted prints /c and /d as missing from the sitemap.

Try it:

1. Print both gaps from this and the previous project side by side.
2. Exclude paginated or filtered URLs from the missing list.
3. Report the share of crawled URLs the sitemap covers.
4. Combine both gaps into one report with two labelled lines.

⚠ Common Mistakes

- This is the mirror of the previous project; subtracting in the wrong direction answers the other question.
- Some crawled URLs (tag pages, filters) are intentionally excluded from the sitemap, so not every gap is a bug.
- Forgetting to sort yields unstable output between runs.

Skills gained set difference | set comprehension | coverage gap

→ Where You'll Use This: Finding crawled pages absent from the sitemap completes the two-way reconciliation, helping ensure valuable pages are declared rather than left to chance discovery.

Mental Model

This is the inventory count run the other way: items on the shelf that the catalogue never listed are the ones you forgot to publish.

Project 90 Histogram the Priority Distribution

The Idea Behind It

A sitemap where every URL claims priority 1.0 sends no useful signal at all. This project bins the priority values with a Counter, then prints a text histogram by multiplying a hash character by each count. Sorting the keys in reverse order draws the bars from highest priority down, revealing at a glance whether the values are meaningfully spread.

What You'll Build & Learn

Build a text histogram of the priority values to check their spread.

- Counter buckets values by frequency
- String multiplication draws a bar
- Uniform priority sends no signal

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 89: find crawled urls missing from the sitemap. Project 91 changes the surface problem while preserving the underlying method.

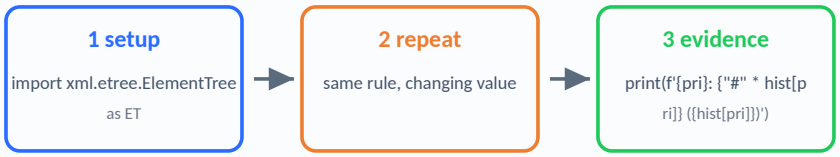
progress check

Reading target: histogram the priority distribution is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{pri}: {"#" * hist[pri]} ({hist[pri]})')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`hist = Counter(u.findtext('priority') for u in root.findall('url'))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{pri}: {"#" * hist[pri]} ({hist[pri]})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `hist = Counter(u.findtext('priority') for u in root.findall('url'))`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Histogram the Priority Distribution: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, collections. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: hist = Counter(u.findtext('priority') for u in root.findall('url')) | Evidence: print(f'{pri}: {"#" * hist[pri]} ({hist[pri]})')

🛠️ Reader to Writer Bridge

1. Read: explain histogram the priority distribution in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 89.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 91 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{pri}: {"#" * hist[pri]} ({hist[pri]})')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of histogram the priority distribution and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `Counter` over the priority strings groups the URLs, and printing one `#` per URL in descending priority order draws a quick text histogram of the distribution.

```
import xml.etree.ElementTree as ET
from collections import Counter
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><priority>1.0</priority></url>'
       '<url><loc>https://example.com/b</loc><priority>0.8</priority></url>'
       '<url><loc>https://example.com/c</loc><priority>0.8</priority></url>'
       '<url><loc>https://example.com/d</loc><priority>0.5</priority></url>'
       '</urlset>')
root = ET.fromstring(xml)
hist = Counter(u.findtext('priority') for u in root.findall('url'))
for pri in sorted(hist, reverse=True):
    print(f'{pri}: {"#" * hist[pri]} ({hist[pri]})')
```

```
output
1.0: # (1)
0.8: ## (2)
0.5: # (1)
```

How it works:

1. Counter tallies the priority text of each url into hist.
2. sorted(hist, reverse=True) orders the buckets high to low.
3. '#' * hist[pri] draws a bar of that length.
4. Each priority prints its bar and its count in parentheses.

Try it:

1. Sort the bars by count instead of by priority value.
2. Normalise bar lengths so the tallest is a fixed width.
3. Flag the sitemap when one priority dominates the distribution.
4. Convert keys to `float` before sorting to handle `0.80` correctly.

⚠ Common Mistakes

- Counting the raw strings means `0.8` and `0.80` would land in different buckets.

- Sorting the string keys works here because they are equal width; mixed widths need float keys.
- A sitemap with every priority at 1.0 produces a single bar, which is the smell this surfaces.

Skills gained Counter | histogram | string multiplication

→ Where You'll Use This: A priority histogram quickly reveals whether a site differentiates its pages or wastes the field by setting everything identically, guiding a priority cleanup.

Mental Model

The histogram is a sound equaliser: if every band peaks at the same level, the mix is flat and tells the listener nothing about emphasis.

Project 91 Histogram the changefreq Values

The Idea Behind It

Knowing how many pages claim daily versus monthly change helps gauge whether your frequencies are realistic. This project tallies the changefreq keywords with a Counter and prints them via most_common, which orders the buckets from most to least frequent. The result is a compact frequency table that exposes overuse of any single keyword.

What You'll Build & Learn

Tally each changefreq keyword and print them ordered by frequency.

- Counter tallies changefreq keywords
- most_common orders by descending count
- Frequency mix reveals unrealistic claims

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 90: histogram the priority distribution. Project 92 changes the surface problem while preserving the underlying method.

progress check

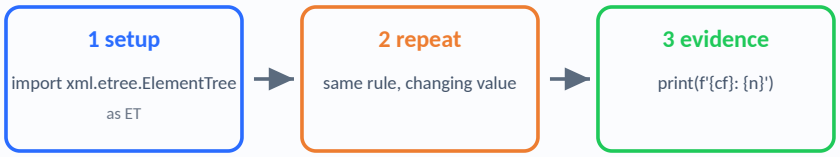
Reading target: histogram the changefreq values is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{cf}: {n}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`hist = Counter(u.findtext('changefreq') for u in root.findall('url'))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{cf}: {n}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `hist = Counter(u.findtext('changefreq') for u in root.findall('url'))`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Histogram the changefreq Values: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, collections. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: hist = Counter(u.findtext('changefreq') for u in root.findall('url')) | Evidence: print(f'{cf}: {n}')

🛠️ Reader to Writer Bridge

1. Read: explain histogram the changefreq values in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 90.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 92 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{cf}: {n}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of histogram the changefreq values and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`Counter.most_common()` returns the frequencies ordered by count, and for equal counts it preserves first-seen insertion order, keeping the output deterministic.

```
import xml.etree.ElementTree as ET
from collections import Counter
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><changefreq>daily</changefreq>
</url>'
       '<url><loc>https://example.com/b</loc><changefreq>daily</changefreq>
</url>'
       '<url><loc>https://example.com/c</loc><changefreq>weekly</changefreq>
</url>'
       '<url><loc>https://example.com/d</loc><changefreq>monthly</changefreq>
</url>'
       '</urlset>')
root = ET.fromstring(xml)
hist = Counter(u.findtext('changefreq') for u in root.findall('url'))
for cf, n in hist.most_common():
    print(f'{cf}: {n}')
```

```
output
daily: 2
weekly: 1
monthly: 1
```

How it works:

1. Four url nodes declare daily twice plus weekly and monthly.
2. Counter tallies each changefreq text into hist.
3. hist.most_common() returns keyword and count pairs sorted by count.
4. The loop prints daily: 2, weekly: 1, monthly: 1.

Try it:

1. Show only the top two most common keywords.
2. Compute the percentage each keyword represents of the total.
3. Warn when one keyword exceeds a share threshold like 80 percent.
4. Add a second weekly URL and watch the ordering shift.

⚠ Common Mistakes

- Ties in `most_common()` fall back to insertion order, which is stable in modern Python but easy to forget.
- A missing `<changefreq>` counts as a `None` key in the histogram.
- An invalid keyword would appear as its own bar; validate separately before counting.

Skills gained `Counter` `most_common` `frequency table`

→ **Where You'll Use This:** A changefreq distribution exposes whether a site over-claims daily updates, which crawlers learn to distrust, undermining the very recrawl hint the tag is meant to give.

⚡ Mental Model

most_common is a vote tally read aloud loudest first: it tells you at once which option swamped the ballot and which barely registered.

Project 92 Flag Non-HTTPS Locations

💡 The Idea Behind It

Modern sites should list only secure URLs, so any loc whose scheme is not https is flagged. This project parses a urlset and applies a startswith check to each loc, catching an ftp scheme and a protocol-relative double-slash that both fail the test. The same prefix technique from the absolute-URL check is reused here to enforce transport security specifically.

What You'll Build & Learn

Flag any loc that does not use the https scheme as insecure.

- startswith enforces the https prefix
- ftp and protocol-relative URLs fail it
- Sitemaps should list only secure URLs

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 91: histogram the changefreq values. Project 93 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag non-https locations is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.

- Evidence: `print(f'{loc}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`verdict = 'secure' if loc.startswith('https://') else 'NOT HTTPS'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'secure' if loc.startswith('https://') else 'NOT HTTPS'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Non-HTTPS Locations: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: verdict = 'secure' if loc.startswith('https://') else 'NOT HTTPS' | Evidence: print(f'{loc}: {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain flag non-https locations in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 91.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 93 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of flag non-https locations and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Testing `startswith('https://')` accepts only the secure scheme; an ftp link or a protocol-relative `//` URL both fail because neither begins with the secure prefix.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>ftp://example.com/file.zip</loc></url>'
       '<url><loc>//example.com/protocol-relative</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    verdict = 'secure' if loc.startswith('https://') else 'NOT HTTPS'
    print(f'{loc}: {verdict}')
```

```
output
https://example.com/a: secure
ftp://example.com/file.zip: NOT HTTPS
//example.com/protocol-relative: NOT HTTPS
```

How it works:

1. Three url nodes use https, ftp, and a protocol-relative prefix.
2. loc.startswith('https://') decides secure versus NOT HTTPS.
3. The ftp and double-slash entries both fail the prefix test.
4. Each loc prints with its verdict.

Try it:

1. Extract and print the actual scheme of each failing loc.
2. Allow http only when an ALLOW_INSECURE flag is set.
3. Count how many locs are insecure across the set.
4. Accept a scheme list by testing membership against `('https://',)`.

⚠ Common Mistakes

- A protocol-relative `//host/path` inherits the page scheme but is not itself secure, so it fails here.
- Matching only the substring `https` (without `://`) would wrongly accept `https-evil.com`.
- Mixed-scheme sitemaps confuse crawlers; pick one canonical scheme site-wide.

Skills gained `startswith` `scheme check` `https enforcement`

→ **Where You'll Use This:** Enforcing https-only locs aligns the sitemap with a secure canonical, avoiding mixed-signal entries that point crawlers at insecure or unresolvable schemes.

⚡ Mental Model

Checking for https is a bouncer at a members-only door: anything arriving by the wrong entrance gets turned back at the threshold.

Project 93 Check Trailing-Slash Consistency

💡 The Idea Behind It

Mixing /page and /page/ in one sitemap can split ranking signals between two URLs that serve the same content. This project collects every loc, counts how many end with a slash using a generator inside sum, derives the remainder, and declares consistency only when one of the two counts is zero. The endswith test cleanly partitions the URLs by trailing-slash style.

What You'll Build & Learn

Count trailing-slash and non-slash URLs and judge whether the sitemap is consistent.

- endswith tests for a trailing slash
- sum over a generator counts matches
- Mixed slash styles split signals

🧭 Skill Roadmap

`Skill step:` moves you from recognition toward independent tracing. It builds on Project 92: flag non-https locations. Project 94 changes the surface problem while preserving the underlying method. *progress check*

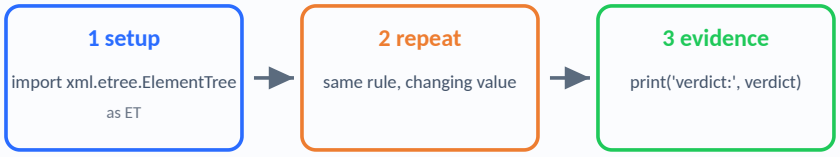
`Reading target:` check trailing-slash consistency is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The comprehension compresses a loop into a readable noun phrase: `make this` for `each item` if `it qualifies`. Expand it mentally when it feels dense.
- `Writing move:` this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.

- Evidence: `print('verdict:', verdict)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`withslash = sum(1 for l in locs if l.endswith('/'))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('verdict:', verdict)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `withslash = sum(1 for l in locs if l.endswith('/'))`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Trailing-Slash Consistency: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: withslash = sum(1 for l in locs if l.endswith('/')) | Evidence: print('verdict:', verdict)

🛠️ Reader to Writer Bridge

1. Read: explain check trailing-slash consistency in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 92.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 94 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('verdict:', verdict)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check trailing-slash consistency and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Counting how many locs end in `/` versus not reveals whether the site sticks to one convention; a mix means the same content may be reachable under two URLs.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/blog/</loc></url>'
       '<url><loc>https://example.com/about</loc></url>'
       '<url><loc>https://example.com/shop/</loc></url>'
       '<url><loc>https://example.com/contact</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
locs = [u.findtext('loc') for u in root.findall('url')]
withslash = sum(1 for l in locs if l.endswith('/'))
without = len(locs) - withslash
print(f'trailing slash: {withslash}, no slash: {without}')
verdict = 'consistent' if withslash == 0 or without == 0 else 'INCONSISTENT'
print('verdict:', verdict)
```

```
output
trailing slash: 2, no slash: 2
verdict: INCONSISTENT
```

How it works:

1. A comprehension collects all four locs.
2. sum with a generator counts those ending in a slash, giving 2.
3. without is the total minus that count, also 2.
4. The verdict is INCONSISTENT because neither count is zero.

Try it:

1. List the URLs of the minority style so they can be normalised.
2. Treat the root URL specially, since the slash there is required.
3. Report which style is the majority to pick a canonical convention.
4. Exclude the bare-domain root before judging consistency.

⚠ Common Mistakes

- The root URL `https://example.com/` legitimately ends in a slash and should not count as inconsistency.
- Trailing-slash style is a convention, not an error, but inconsistency can dilute ranking signals.
- `endswith('/')` treats a query-string URL ending in `/` the same as a path, which may mislead.

Skills gained endswith | sum generator | consistency check

→ Where You'll Use This: Trailing-slash consistency prevents the sitemap from declaring duplicate variants of the same page, a subtle dilution that fragments link and ranking signals.

⚡ Mental Model

Trailing slashes are like two spellings of one name on a guest list: the host may seat the same person twice and split the credit.

Project 94 Count Image and Video Alternates

💡 The Idea Behind It

Image and video sitemaps nest image and video tags inside each url, and this project counts them per page. It parses the urlset and for every url calls findall on both image and video, taking the length of each result. Counting nested children rather than reading their contents gives a quick coverage tally of rich media per page.

What You'll Build & Learn

Count the image and video child elements nested under each URL.

- image and video tags nest inside url
- findall plus len counts nested children
- Rich-media counts audit coverage

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 93: check trailing-slash consistency. Project 95 changes the surface problem while preserving the underlying method.

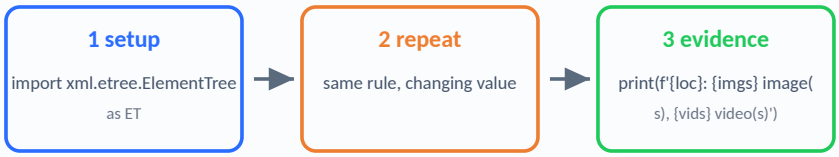
progress check

Reading target: count image and video alternates is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {imgs} image(s), {vids} video(s)')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for u in root.findall('url'):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{loc}: {imgs} image(s), {vids} video(s)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for u in root.findall('url'):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Image and Video Alternates: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: for u in root.findall('url'): | Evidence: print(f'{loc}: {imgs} image(s), {vids} video(s)')

🛠️ Reader to Writer Bridge

1. Read: explain count image and video alternates in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 93.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 95 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {imgs} image(s), {vids} video(s)')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count image and video alternates and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`findall('image')` and `findall('video')` count the media child elements within each `<url>`; real media sitemaps namespace these tags but the counting logic is the same.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/photo</loc>'
       '<image><loc>https://example.com/a.jpg</loc></image>'
       '<image><loc>https://example.com/b.jpg</loc></image></url>'
       '<url><loc>https://example.com/clip</loc>'
```

```
        '<video><loc>https://example.com/v.mp4</loc></video></url>'
        '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    imgs = len(u.findall('image'))
    vids = len(u.findall('video'))
    print(f'{loc}: {imgs} image(s), {vids} video(s)')
```

```
output
https://example.com/photo: 2 image(s), 0 video(s)
https://example.com/clip: 0 image(s), 1 video(s)
```

How it works:

1. The first url holds two image children, the second one video child.
2. u.findall('image') and u.findall('video') gather the nested tags.
3. len of each result gives the per-page counts.
4. Each loc prints with its image and video totals.

Try it:

1. Also extract and print each nested media loc.
2. Flag pages that declare zero media when media is expected.
3. Total all images and videos across the whole sitemap.
4. Sum the image counts across all URLs for a site-wide total.

⚠ Common Mistakes

- Real image sitemaps use the `image:` namespace prefix, so a live file needs the namespaced tag name.
- An `<image>` block has its own nested `<loc>`, distinct from the page `<loc>`; do not confuse them.
- A page may legitimately have zero media, so absence is not an error.

Skills gained findall | nested elements | media audit

→ Where You'll Use This: Counting media alternates audits whether image and video sitemaps adequately cover a site's rich assets, which influences eligibility for visual search features.

⚡ Mental Model

Each url is a photo album page: counting the image and video tags is flipping it open to see how many prints are slotted in.

Project 95 Count Hreflang Alternates per URL

The Idea Behind It

International sites attach link rel=alternate hreflang tags so search engines serve the right language version. This project parses each url, gathers its link children with findall, and reads the hreflang attribute from each via the get method. Counting and listing the language codes per URL audits how thoroughly localisation alternates are declared.

What You'll Build & Learn

Count and list the hreflang alternates declared under each URL.

- link elements carry an hreflang attribute
- get reads an attribute value
- Counting alternates audits localisation

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 94: count image and video alternates. Project 96 changes the surface problem while preserving the underlying method.

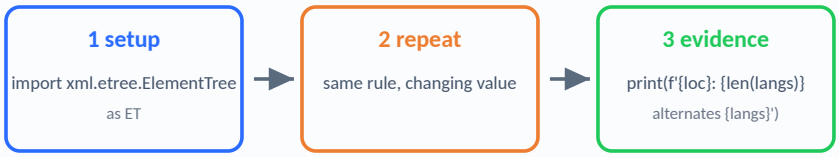
progress check

Reading target: count hreflang alternates per url is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {len(langs)} alternates {langs}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for u in root.findall('url'):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{loc}: {len(langs)} alternates {langs}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for u in root.findall('url'):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Hreflang Alternates per URL: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: for u in root.findall('url'): | Evidence: print(f'{loc}: {len(langs)} alternates {langs}')

🛠️ Reader to Writer Bridge

1. Read: explain count hreflang alternates per url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 94.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 96 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {len(langs)} alternates {langs}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count hreflang alternates per url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`findall('link')` collects the alternate-language link elements and `.get('hreflang')` reads each language code from its attribute; live sitemaps use the `xhtml:` namespace but attribute access is identical.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/en/page</loc>'
       '<link rel="alternate" hreflang="en"
href="https://example.com/en/page"/>'
       '<link rel="alternate" hreflang="de"
href="https://example.com/de/page"/>'
```

```
        '<link rel="alternate" hreflang="fr"
href="https://example.com/fr/page"/></url>'
        '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    langs = [lk.get('hreflang') for lk in u.findall('link')]
    print(f'{loc}: {len(langs)} alternates {langs}')
```

```
output
https://example.com/en/page: 3 alternates ['en', 'de', 'fr']
```

How it works:

1. One url holds three link children for en, de, and fr.
2. u.findall('link') gathers the alternate link elements.
3. lk.get('hreflang') reads each language code into a list.
4. The loc prints with the count and the list of codes.

Try it:

1. Flag URLs missing a self-referential hreflang for their own language.
2. Detect a duplicate language code declared twice on one URL.
3. Also read and print each alternate's href attribute.
4. Detect a missing `x-default` by checking the hreflang values.

⚠ Common Mistakes

- Hreflang links live in the `xhtml` namespace in real sitemaps; the fixture drops it so the bare tag matches.
- A complete hreflang set should also include an `x-default`; counting alternates does not check that.
- Reciprocity matters: each alternate page must link back, which this single-URL view cannot verify.

Skills gained findall | attribute get | hreflang

→ Where You'll Use This: Auditing hreflang alternates per URL helps catch incomplete or asymmetric language clusters, a frequent cause of wrong-locale pages ranking in international search.

Mental Model

hreflang tags are translated road signs at a border: each one points a traveller to the version of the page written in their language.

Project 96 Read a News Sitemap Block

The Idea Behind It

Google News sitemaps nest a news block with a title and publication date inside each url. This project parses the urlset and uses find rather than findall to test whether each url has a news child, branching to print either the headline and date or a plain note. The None check distinguishes news-eligible pages from ordinary ones.

What You'll Build & Learn

Read the news title and date from each URL that carries a news block.

- find returns one child or None
- A None check detects the news block
- news nests a title and pubdate

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 95: count hreflang alternates per url. Project 97 changes the surface problem while preserving the underlying method.

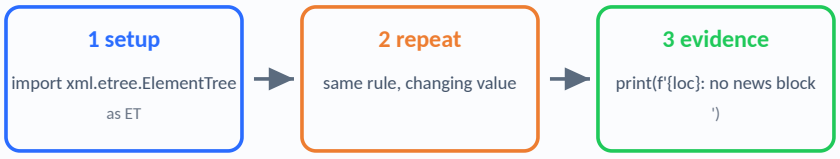
progress check

Reading target: read a news sitemap block is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: no news block')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`if news is not None:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: no news block')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if news is not None:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Read a News Sitemap Block: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: if news is not None: | Evidence: print(f'{loc}: no news block')

Reader to Writer Bridge

1. Read: explain read a news sitemap block in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 95.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 97 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: no news block')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of read a news sitemap block and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

From Concept to Code (Python)

`u.find('news')` returns the nested news element or `None`, and reading its child `<title>`/`<pubdate>` pulls the news metadata; live News sitemaps namespace these tags identically in structure.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/news</loc>'
       '<news><title>Big Story</title><pubdate>2026-04-01</pubdate></news>
</url>'
       '<url><loc>https://example.com/plain</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    news = u.find('news')
    if news is not None:
        print(f'{loc}: NEWS "{news.findtext("title")}"
({news.findtext("pubdate")})')
    else:
        print(f'{loc}: no news block')
```

```
output
https://example.com/news: NEWS "Big Story" (2026-04-01)
https://example.com/plain: no news block
```

How it works:

1. One url holds a news block, the other does not.
2. u.find('news') returns the element or None.
3. When present, findtext reads the title and pubdate inside it.
4. Each loc prints either the news details or no news block.

Try it:

1. Flag a news block missing its title or pubdate child.
2. Validate the pubdate parses as a real date.
3. Count how many URLs carry a news block versus none.
4. Validate that `pubdate` parses as an ISO date.

⚠ Common Mistakes

- A real news sitemap uses the `news:` namespace, so the tag names need a prefix on live data.
- News sitemaps should only contain articles from the last two days; older entries are ignored.
- `find` returning `None` must be guarded before calling `.findtext` on it.

Skills gained find | None check | news sitemap

→ Where You'll Use This: Extracting news blocks audits which pages are declared news-eligible, important for publishers chasing Top Stories placement where the news namespace is required.

⚡ Mental Model

find is peeking into one drawer rather than the whole dresser: it either hands you the news folder or comes back empty, and you act accordingly.

Project 97 Measure Sitemap Byte Size

💡 The Idea Behind It

Besides the fifty-thousand-URL cap, an uncompressed sitemap may not exceed fifty megabytes. This project encodes the urlset string to UTF-8 bytes, measures that length, and compares it to a MAX_BYTES constant computed as 50 times 1024 times 1024. Encoding before measuring matters because byte length, not character count, is what the limit governs.

What You'll Build & Learn

Measure the sitemap's encoded byte size and check it against the 50 MB limit.

- encode turns text into bytes
- len of bytes is the true file size

- Sitemaps are capped at 50 MB

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 96: read a news sitemap block. Project 98 changes the surface problem while preserving the underlying method. *progress check*

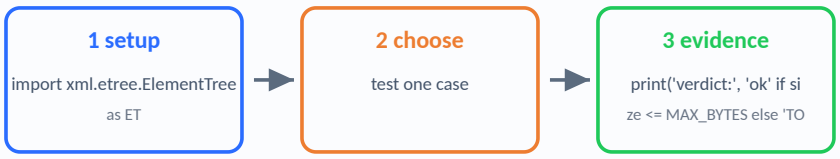
Reading target: measure sitemap byte size is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it')`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Measure Sitemap Byte Size: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it') | Evidence: print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it')

Reader to Writer Bridge

1. Read: explain measure sitemap byte size in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 96.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 98 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure sitemap byte size and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Encoding the XML to UTF-8 bytes and comparing against `50 * 1024 * 1024` enforces the 50 MB uncompressed-size limit, the constraint that bites before the URL count on link-heavy pages.

```
import xml.etree.ElementTree as ET
MAX_BYTES = 50 * 1024 * 1024
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '</urlset>')
size = len(xml.encode('utf-8'))
print(f'sitemap size: {size} bytes (limit {MAX_BYTES})')
print('verdict:', 'ok' if size <= MAX_BYTES else 'TOO LARGE, split it')
```

```
output
sitemap size: 103 bytes (limit 52428800)
verdict: ok
```

How it works:

1. MAX_BYTES is 50 times 1024 times 1024, the byte limit.
2. xml.encode('utf-8') converts the string to bytes.
3. len of those bytes gives 103, the encoded size.
4. The verdict compares size to MAX_BYTES and prints ok.

Try it:

1. Estimate the size after gzip compression for the compressed limit.
2. Lower MAX_BYTES to force the TOO LARGE branch.
3. Print the size in megabytes alongside the raw byte count.
4. Switch the unit to mebibytes in the printout by dividing the size by 1024*1024.

⚠ Common Mistakes

- `len(xml)` counts characters, not bytes; multibyte content needs `.encode('utf-8')` for a true size.
- The 50 MB limit is on the uncompressed file even though you may serve it gzipped.
- Both the URL count and the byte size must pass; either one can force a split.

Skills gained encode | byte length | size limit

→ Where You'll Use This: The byte-size limit, separate from the URL count, forces a split when long URLs or rich metadata bloat a file, so this check guards the second sitemap ceiling.

Mental Model

Byte size is the weight on a luggage scale: even a bag under the item count can tip past the kilo limit and force you to repack.

Project 98 Apply the Default Priority of 0.5

The Idea Behind It

When priority is omitted, crawlers assume a default of 0.5, and this project fills that gap explicitly. It reads each url's priority, casts it to float when present, and substitutes DEFAULT when findtext returns None, also tagging which value was supplied versus inferred. Normalising every URL to a numeric priority lets later analysis treat the whole set uniformly.

What You'll Build & Learn

Assign the default priority of 0.5 to any URL that omits the tag.

- Omitted priority defaults to 0.5
- findtext None triggers the default
- Normalising fills gaps for analysis

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 97: measure sitemap byte size. Project 99 changes the surface problem while preserving the underlying method. *progress check*

Reading target: apply the default priority of 0.5 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {pri}{note}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`pri = float(raw) if raw is not None else DEFAULT`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: {pri}{note}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `pri = float(raw) if raw is not None else DEFAULT`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Apply the Default Priority of 0.5: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: pri = float(raw) if raw is not None else DEFAULT | Evidence: print(f'{loc}: {pri}{note}')

🛠️ Reader to Writer Bridge

1. Read: explain apply the default priority of 0.5 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 97.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 99 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {pri}{note}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of apply the default priority of 0.5 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Treating a missing `<priority>` as the protocol default of 0.5 means an explicit 0.5 and an omitted tag are scored the same, exactly as a crawler would interpret them.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><priority>0.8</priority></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '<url><loc>https://example.com/c</loc><priority>0.5</priority></url>'
       '</urlset>')
root = ET.fromstring(xml)
DEFAULT = 0.5
for u in root.findall('url'):
    loc = u.findtext('loc')
    raw = u.findtext('priority')
    pri = float(raw) if raw is not None else DEFAULT
    note = '' if raw is not None else ' (default)'
    print(f'{loc}: {pri}{note}')
```

```
output
https://example.com/a: 0.8
https://example.com/b: 0.5 (default)
https://example.com/c: 0.5
```

How it works:

1. Three url nodes give 0.8, no priority, and 0.5.
2. raw is the priority text or None.
3. pri is float(raw) when present, otherwise the DEFAULT of 0.5.
4. The middle URL prints 0.5 with a (default) note.

Try it:

1. Count how many URLs relied on the default.
2. Feed the normalised priorities into a histogram from earlier.
3. Make the default configurable rather than hard-coded.
4. Change `DEFAULT` to 0.0 and watch URL b drop.

⚠ Common Mistakes

- `float(raw)` crashes when `raw` is `None`, so the default must be applied before conversion.
- An explicit `0.5` is indistinguishable from a default unless you track the `note` separately.

- Priority is only a relative hint; the default rarely changes crawl outcomes.

Skills gained default value | float casting | None check

→ Where You'll Use This: Filling the implied 0.5 default lets a histogram or comparison treat declared and undeclared URLs on equal footing, avoiding skew from missing tags.

Mental Model

Filling the default is setting a thermostat's resting temperature: rooms nobody adjusted still hold a known value instead of reading blank.

Project 99 Sum URL Counts Across an Index

The Idea Behind It

A sitemap index hides the real URL total, so you must add up each child's count. This project parses a sitemapindex, looks up a precomputed count for every child loc in a dictionary, and sums those numbers to report the site's full declared coverage. The dict lookup keyed on each child URL stands in for fetching and counting the real files.

What You'll Build & Learn

Sum the URL counts of every child sitemap to report total coverage.

- An index hides the true URL total
- A dict maps each child to its count
- sum aggregates across the children

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 98: apply the default priority of 0.5. Project 100 changes the surface problem while preserving the underlying method.

progress check

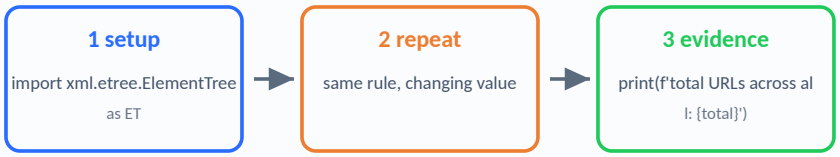
Reading target: sum url counts across an index is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'total URLs across all: {total}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`total = sum(child_counts[sm.findtext('loc')] for sm in root.findall('sitemap'))`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'total URLs across all: {total}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `total = sum(child_counts[sm.findtext('loc')] for sm in root.findall('sitemap'))`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Sum URL Counts Across an Index: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, sitemapindex. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: total = sum(child_counts[sm.findtext('loc')] for sm in root.findall('si... | Evidence: print(f'total URLs across all: {total}')

🛠️ Reader to Writer Bridge

1. Read: explain sum url counts across an index in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 98.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 100 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'total URLs across all: {total}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of sum url counts across an index and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Looking up each child sitemap's known URL count by its loc and summing them aggregates the index into one site-wide total, which can far exceed any single file's 50,000 cap.

```
import xml.etree.ElementTree as ET
xml = ('<sitemapindex>'
       '<sitemap><loc>https://example.com/sm-1.xml</loc></sitemap>'
       '<sitemap><loc>https://example.com/sm-2.xml</loc></sitemap>'
       '<sitemap><loc>https://example.com/sm-3.xml</loc></sitemap>'
       '</sitemapindex>')
child_counts = {
    'https://example.com/sm-1.xml': 50000,
    'https://example.com/sm-2.xml': 50000,
    'https://example.com/sm-3.xml': 12000,
}
root = ET.fromstring(xml)
total = sum(child_counts[sm.findtext('loc')] for sm in root.findall('sitemap'))
print(f'index references {len(root.findall("sitemap"))} sitemaps')
print(f'total URLs across all: {total}')
```

```
output
index references 3 sitemaps
total URLs across all: 112000
```

How it works:

1. child_counts maps each child loc to its URL count.
2. The index parses into three sitemap nodes.
3. sum looks up each child's count via its loc and adds them.
4. The totals 50000 plus 50000 plus 12000 print as 112000.

Try it:

1. Handle a child loc missing from the dict with a default of zero.
2. Flag any single child whose count exceeds the 50000 cap.
3. Print the average URLs per child sitemap.
4. Use `.get(loc, 0)` so an unlisted child counts as zero rather than crashing.

⚠ Common Mistakes

- A missing key in `child_counts` raises `KeyError`; default with `.get(loc, 0)` if children may be absent.
- An index itself caps at 50,000 child sitemaps, giving a theoretical ceiling of 2.5 billion URLs.

- Trusting declared counts assumes each child actually contains that many URLs.

Skills gained dict lookup | sum | index aggregation

→ Where You'll Use This: Summing per-child counts reconstructs a sharded site's true declared coverage, the number you compare against Search Console's indexed total for big properties.

⚡ Mental Model

The index is a stack of sealed boxes labelled with counts: you add the labels to know the warehouse total without opening a single box.

Project 100 Detect Case-Collision loc Variants

💡 The Idea Behind It

URLs differing only by letter case can resolve to the same page on a case-insensitive server, creating hidden duplicates. This project groups locs by their lowercased form using a dict with setdefault, appending each original into its bucket. Any bucket holding more than one original reveals a case collision, which the final loop prints.

What You'll Build & Learn

Group locs by lowercased form to surface URLs that collide only on case.

- Lowercasing normalises case differences
- setdefault groups values into buckets
- Buckets with two members are collisions

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 99: sum url counts across an index. Project 101 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect case-collision loc variants is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

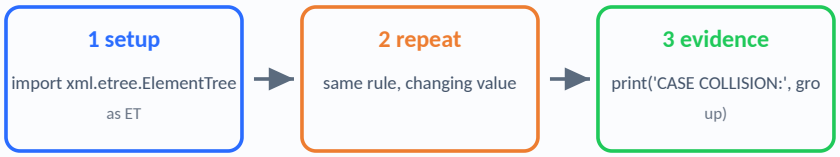
🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size

limits and errors.

- Evidence: `print('CASE COLLISION:', group)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`if len(group) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('CASE COLLISION:', group)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(group) > 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Case-Collision loc Variants: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: if len(group) > 1: | Evidence: print('CASE COLLISION:', group)

🛠️ Reader to Writer Bridge

1. Read: explain detect case-collision loc variants in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 99.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 101 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('CASE COLLISION:', group)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect case-collision loc variants and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Bucketing each loc under its lowercased key groups case variants together; any bucket with more than one member is a potential duplicate on a case-insensitive host.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/A</loc></url>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/B</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
seen = {}
for u in root.findall('url'):
    loc = u.findtext('loc')
    key = loc.lower()
    seen.setdefault(key, []).append(loc)
for key, group in sorted(seen.items()):
    if len(group) > 1:
        print('CASE COLLISION:', group)
```

```
output
CASE COLLISION: ['https://example.com/A', 'https://example.com/a']
```

How it works:

1. Three locs include /A and /a plus a distinct /B.
2. key is each loc lowercased.
3. setdefault(key, []).append(loc) groups originals by lowercase form.
4. Buckets longer than one print as a CASE COLLISION.

Try it:

1. Also collapse trailing slashes before grouping for stricter matching.
2. Print the count of collision groups found.
3. Pick one canonical casing per group to keep.
4. Lowercase only the host, leaving the path untouched, to avoid false positives.

⚠ Common Mistakes

- Lowercasing the whole URL is risky: the path may be genuinely case-sensitive on some servers.
- `setdefault` builds the lists in one pass; a plain dict assignment would overwrite earlier entries.
- Sorting the dict items keeps the report stable across runs.

Skills gained setdefault lowercasing grouping

→ Where You'll Use This: Case-collision detection catches duplicate URLs introduced by inconsistent casing in CMS exports, which can dilute signals or trigger duplicate-content handling.

Mental Model

Lowercasing before grouping is filing names alphabetically while ignoring capital letters: McLaughlin and mclaughlin land in the same folder.

Project 101 Reject Cross-Domain Locations

The Idea Behind It

Every loc in a sitemap must belong to the host that serves it, since cross-domain URLs are ignored. This project extracts the host from each loc by splitting on the scheme separator and then the first slash, comparing it to an ALLOWED_HOST constant. The double split isolates the netloc without a full URL parser, flagging strays from other domains.

What You'll Build & Learn

Reject any loc whose host differs from the sitemap's allowed domain.

- Splitting on :// and / isolates the host
- Compare the host to an allowed value
- Cross-domain locs are ignored by crawlers

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 100: detect case-collision loc variants. Project 102 changes the surface problem while preserving the underlying method.

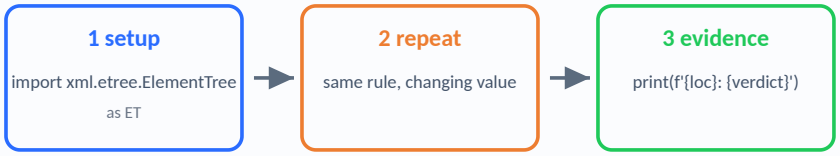
progress check

Reading target: reject cross-domain locations is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`verdict = 'ok' if host == ALLOWED_HOST else f'CROSS-DOMAIN ({host})'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{loc}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'ok' if host == ALLOWED_HOST else f'CROSS-DOMAIN ({host})'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Reject Cross-Domain Locations: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: verdict = 'ok' if host == ALLOWED_HOST else f'CROSS-DOMAIN ({host})' | Evidence: print(f'{loc}: {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain reject cross-domain locations in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 100.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 102 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of reject cross-domain locations and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Splitting the loc on `://` then on the first `/` isolates the host, and comparing it to the allowed host flags any URL the sitemap is not authorised to declare.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://shop.example/a</loc></url>'
       '<url><loc>https://other.example/b</loc></url>'
       '<url><loc>https://shop.example/c</loc></url>'
       '</urlset>')
```

```
ALLOWED_HOST = 'shop.example'
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    host = loc.split('://', 1)[1].split('/', 1)[0]
    verdict = 'ok' if host == ALLOWED_HOST else f'CROSS-DOMAIN ({host})'
    print(f'{loc}: {verdict}')
```

```
output
https://shop.example/a: ok
https://other.example/b: CROSS-DOMAIN (other.example)
https://shop.example/c: ok
```

How it works:

1. Three locs span shop.example and one other.example.
2. loc.split('://', 1)[1] drops the scheme.
3. A second split on the first slash yields the host.
4. host equal to ALLOWED_HOST prints ok, otherwise CROSS-DOMAIN.

Try it:

1. Use urllib.parse.urlparse to read the host more robustly.
2. Allow a set of approved subdomains rather than one host.
3. Strip a port number from the host before comparing.
4. Allow a set of hosts instead of a single one.

⚠ Common Mistakes

- Splitting on `/` without first removing the scheme would grab `https:` as the host.
- A `www.` versus bare-domain mismatch counts as cross-domain unless you normalise it.
- Sitemaps in a Search Console domain property may legitimately span subdomains; adjust the rule accordingly.

Skills gained string split | host extraction | domain check

→ Where You'll Use This: Host validation catches stray cross-domain URLs that slip into a shared sitemap template, which search engines discard, silently shrinking declared coverage.

⚡ Mental Model

Checking the host is a postcode filter on a local mailing: any letter addressed to another town gets pulled before the round begins.

Project 102 Split a Urlset Into Capped Chunks

The Idea Behind It

When a urlset grows too big, you slice it into files of at most N URLs each. This project parses the urlset, collects the locs into a list, and uses a range-stepped comprehension to cut that list into chunks of PER_FILE entries. Each chunk corresponds to one output sitemap, and the loop prints how the split would lay out.

What You'll Build & Learn

Chunk a list of URLs into capped slices, one per output sitemap file.

- Slicing with a step builds equal chunks
- range steps by the per-file cap
- Each chunk maps to one output file

Skill Roadmap

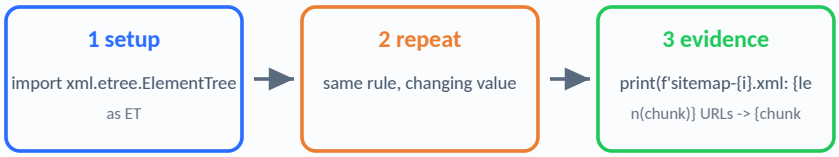
Skill step: pushes the same idea into a less automatic situation. It builds on Project 101: reject cross-domain locations. Project 103 changes the surface problem while preserving the underlying method. *progress check*

Reading target: split a urlset into capped chunks is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'sitemap-{i}.xml: {len(chunk)} URLs -> {chunk}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`locs = [u.findtext('loc') for u in root.findall('url')]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'sitemap-{i}.xml: {len(chunk)} URLs -> {chunk}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `locs = [u.findtext('loc') for u in root.findall('url')]`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Split a Urlset Into Capped Chunks: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: locs = [u.findtext('loc') for u in root.findall('url')] | Evidence: print(f'sitemap-{i}.xml: {len(chunk)} URLs -> {chunk}')

🛠️ Reader to Writer Bridge

1. Read: explain split a urlset into capped chunks in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 101.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 103 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'sitemap-{i}.xml: {len(chunk)} URLs -> {chunk}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split a urlset into capped chunks and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Slicing the loc list in steps of `PER_FILE` produces evenly sized chunks (with a smaller final one), the layout you would write into a multi-file sitemap set under an index.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '<url><loc>https://example.com/c</loc></url>'
       '</urlset>')
```

```
root = ET.fromstring(xml)
PER_FILE = 2
locs = [u.findtext('loc') for u in root.findall('url')]
chunks = [locs[i:i + PER_FILE] for i in range(0, len(locs), PER_FILE)]
for i, chunk in enumerate(chunks, 1):
    print(f'sitemap-{i}.xml: {len(chunk)} URLs -> {chunk}')
```

```
output
sitemap-1.xml: 2 URLs -> ['https://example.com/a', 'https://example.com/b']
sitemap-2.xml: 1 URLs -> ['https://example.com/c']
```

How it works:

1. locs collects the three URLs from the urlset.
2. PER_FILE is 2, the chunk size.
3. The comprehension slices locs[i:i+2] for i stepping by 2.
4. Two chunks print as sitemap-1 with two URLs and sitemap-2 with one.

Try it:

1. Set PER_FILE to 50000 to match the real cap.
2. Wrap each chunk into a full urlset string.
3. Print the suggested index entry for each generated file.
4. Raise `PER_FILE` to 3 and confirm a single chunk results.

⚠ Common Mistakes

- Using `PER_FILE` of 0 would make `range` raise; the cap must be positive.
- The last chunk is usually smaller than the cap, which is expected, not an error.
- In production `PER_FILE` is 50000, but a small value makes the splitting visible here.

Skills gained list slicing | range step | chunking

→ Where You'll Use This: Chunking is the mechanism behind splitting an over-cap sitemap into compliant shards, the operational counterpart to the index file that ties them together.

Mental Model

Chunking is portioning a big batch of dough into trays: each tray holds the same maximum and you bake them as separate loaves.

Project 103 Reject Impossible Calendar Dates

The Idea Behind It

A date like 2026-02-30 matches the digit pattern yet is not a real day, and strptime rejects it where a regex would wave it through. This project parses each lastmod with the ISO format and catches the ValueError that impossible months and days raise. It shows why semantic date parsing beats shape-only validation for calendar correctness.

What You'll Build & Learn

Reject lastmod values that fit the digit pattern but name impossible calendar dates.

- Pattern shape does not guarantee a real date
- strptime validates calendar correctness
- ValueError flags impossible days

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 102: split a urlset into capped chunks. Project 104 changes the surface problem while preserving the underlying method.

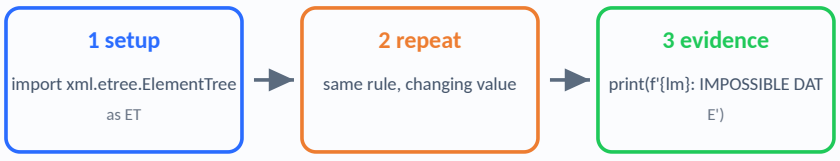
progress check

Reading target: reject impossible calendar dates is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{lm}: IMPOSSIBLE DATE')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`for u in root.findall('url'):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{lm}: IMPOSSIBLE DATE')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for u in root.findall('url'):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Read the try path as the hoped-for route and the catch/except/finally path as the boundary plan.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Reject Impossible Calendar Dates: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, datetime. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: for u in root.findall('url'): | Evidence: print(f'{lm}: IMPOSSIBLE DATE')

🛠️ Reader to Writer Bridge

1. Read: explain reject impossible calendar dates in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 102.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 104 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{lm}: IMPOSSIBLE DATE')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of reject impossible calendar dates and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`strptime` does real calendar validation, so month 13 and February 30 raise `ValueError` even though they fit the `YYYY-MM-DD` shape; a naive regex would accept both.

```
import xml.etree.ElementTree as ET
from datetime import datetime
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><lastmod>2026-13-01</lastmod>
</url>'
       '<url><loc>https://example.com/b</loc><lastmod>2026-02-30</lastmod>
</url>'
       '<url><loc>https://example.com/c</loc><lastmod>2026-02-28</lastmod>
</url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    lm = u.findtext('lastmod')
    try:
        datetime.strptime(lm, '%Y-%m-%d')
        print(f'{lm}: valid calendar date')
    except ValueError:
        print(f'{lm}: IMPOSSIBLE DATE')
```

```
output
```

```
2026-13-01: IMPOSSIBLE DATE
2026-02-30: IMPOSSIBLE DATE
2026-02-28: valid calendar date
```

How it works:

1. Three lastmod values include month 13, February 30, and February 28.
2. strptime tries to parse each as YYYY-MM-DD.
3. The month-13 and Feb-30 values raise ValueError.
4. Only 2026-02-28 prints as a valid calendar date.

Try it:

1. Add a leap-day test with 2024-02-29 versus 2026-02-29.
2. Report which component, month or day, made a date impossible.
3. Collect only the impossible dates into a list.
4. Add `2026-02-29` and confirm it fails since 2026 is not a leap year.

⚠ Common Mistakes

- A regex that only checks digit groups passes `2026-13-01`; only true date parsing catches it.
- Leap-year rules mean `2024-02-29` is valid but `2026-02-29` is not, which strptime handles for you.
- `strptime` parses a fixed string and does not consult the current date, keeping the test deterministic.

Skills gained strptime | calendar validation | try/except

→ Where You'll Use This: Calendar validation catches dates that pass a naive regex but are genuinely impossible, preventing nonsense lastmod values that crawlers cannot interpret.

⚡ Mental Model

strptime is a calendar with real page tear-offs: ask for February 30 and there simply is no leaf to flip to, so it refuses.

Project 104 Split a lastmod Date From Its Time

💡 The Idea Behind It

A lastmod may be a plain date or a full timestamp carrying a T separator before the time. This project splits each value on T, keeps the part before it as the date, and tests for the separator's presence to note whether a time was attached. The split isolates the date for comparison while flagging which entries include a time component.

What You'll Build & Learn

Separate the date portion of each lastmod from any time component.

- T separates the date from the time
- split on T isolates the date part
- Membership tests for a time component

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 103: reject impossible calendar dates. Project 105 changes the surface problem while preserving the underlying method.

progress check

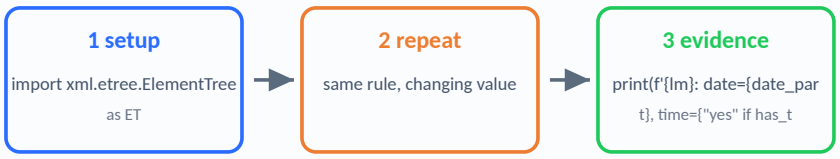
Reading target: split a lastmod date from its time is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build

Move	Code to inspect	Why it matters
		and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Split a lastmod Date From Its Time: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}') | Evidence: print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}')

🛠️ Reader to Writer Bridge

1. Read: explain split a lastmod date from its time in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 103.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 105 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split a lastmod date from its time and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Splitting on the ISO `T` separator (with a limit of 1) returns the date portion regardless of whether a time follows, letting date-level logic ignore the optional clock component.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><lastmod>2026-05-
01T14:30:00+00:00</lastmod></url>'
       '<url><loc>https://example.com/b</loc><lastmod>2026-05-02</lastmod>
</url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    lm = u.findtext('lastmod')
    date_part = lm.split('T', 1)[0]
    has_time = 'T' in lm
    print(f'{lm}: date={date_part}, time={"yes" if has_time else "no"}')
```

```
output
2026-05-01T14:30:00+00:00: date=2026-05-01, time=yes
2026-05-02: date=2026-05-02, time=no
```

How it works:

1. One lastmod is a full timestamp, the other a plain date.
2. lm.split('T', 1)[0] keeps the date portion.
3. 'T' in lm reports whether a time is present.
4. Each value prints its date part and a yes or no for time.

Try it:

1. Also extract and print the time portion when present.
2. Strip the timezone offset from the time component.
3. Validate the date part with strptime after splitting.
4. Parse the full timestamp with `datetime.fromisoformat` and print it.

⚠ Common Mistakes

- A lowercase `t` would not split; ISO-8601 uses an uppercase `T` separator.
- The time zone offset after the time is dropped here; preserve it if you need exact ordering.
- Comparing full timestamps as strings only works when their formats match exactly.

Skills gained string split | membership test | timestamp parsing

→ Where You'll Use This: Separating date from time normalises mixed lastmod formats so day-level comparisons and staleness checks work uniformly regardless of timestamp precision.

⚡ Mental Model

The T separator is the comma in a date-and-time stamp: split there and you hold the day in one hand, the clock reading in the other.

Project 105 Trim Whitespace Around a loc

💡 The Idea Behind It

Hand-edited sitemaps sometimes pad a loc with spaces or newlines, which quietly break URL comparisons. This project reads each loc, strips surrounding whitespace, and flags any value that changed once trimmed. Comparing the raw text to its stripped form both normalises the URL and identifies which entries were dirty in the first place.

What You'll Build & Learn

Strip whitespace from each loc and flag the entries that had padding.

- strip removes surrounding whitespace
- Comparing raw to stripped detects padding

- Padded locs break comparisons

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 104: split a lastmod date from its time. Project 106 changes the surface problem while preserving the underlying method.

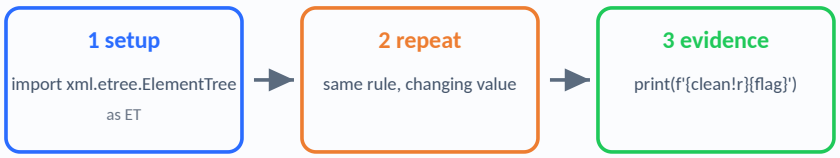
progress check

Reading target: trim whitespace around a loc is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{clean!r}{flag}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`flag = '' if raw == clean else ' (had whitespace)'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'{clean!r}{flag}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = '' if raw == clean else ' (had whitespace)'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Trim Whitespace Around a loc: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: flag = '' if raw == clean else ' (had whitespace)' | Evidence: print(f'{clean!r}{flag}')

Reader to Writer Bridge

1. Read: explain trim whitespace around a loc in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 104.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 106 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{clean!r}{flag}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of trim whitespace around a loc and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`str.strip()` removes leading and trailing whitespace including newlines; comparing the stripped value to the original flags entries that carried stray padding.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>   https://example.com/b   </loc></url>'
       '<url><loc>https://example.com/c\n</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    raw = u.findtext('loc')
    clean = raw.strip()
    flag = '' if raw == clean else ' (had whitespace)'
    print(f'{clean!r}{flag}')
```

```
output
'https://example.com/a'
'https://example.com/b' (had whitespace)
'https://example.com/c' (had whitespace)
```

How it works:

1. Three locs include one clean, one space-padded, and one with a newline.
2. raw holds the original text; clean is raw.strip().
3. raw not equal to clean flags the padded entries.
4. Each cleaned loc prints in repr form with a whitespace note.

Try it:

1. Also collapse internal whitespace inside the URL.
2. Count how many locs needed trimming.
3. Feed the cleaned locs into the duplicate detector from earlier.
4. Strip only newlines with `strip(chr(10))` to preserve intentional spaces.

⚠ Common Mistakes

- Unstripped locs cause set comparisons and dedup checks to treat ` url ` and `url` as different.
- `strip()` only trims the ends; internal whitespace inside a URL is a different (encoding) problem.
- Whitespace can be a literal space, tab, or newline; `strip()` handles all of them.

Skills gained strip repr normalisation

→ Where You'll Use This: Whitespace trimming normalises locs before deduplication or crawl comparison, since an invisible trailing newline otherwise makes two identical URLs look distinct.

Mental Model

Stripping a loc is trimming the frayed edges off a label so the barcode scans: the padding was invisible but kept the reader from matching it.

Project 106 Spot Sitemap URLs Blocked by Robots

The Idea Behind It

Listing a URL in the sitemap while robots.txt disallows its path sends crawlers a contradictory message. This project extracts each loc's path by splitting off the scheme and host, then tests whether that path starts with any disallowed prefix using any over a tuple. Matching paths against the disallow rules surfaces the conflict between inviting and blocking the same page.

What You'll Build & Learn

Detect sitemap URLs whose path is blocked by robots.txt disallow rules.

- Splitting isolates the URL path
- any tests against several prefixes
- Listing a blocked path is contradictory

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 105: trim whitespace around a loc. Project 107 changes the surface problem while preserving the underlying method.

progress check

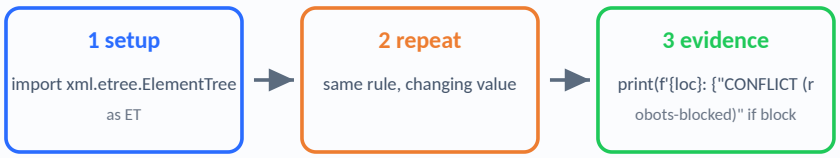
Reading target: spot sitemap urls blocked by robots is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {"CONFLICT (robots-blocked)" if blocked else "ok"}')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`path = '/' + host_path.split('/', 1)[1] if '/' in host_path else '/'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: {"CONFLICT (robots-blocked)" if blocked else "ok"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `path = '/' + host_path.split('/', 1)[1] if '/' in host_path else '/'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Spot Sitemap URLs Blocked by Robots: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: path = '/' + host_path.split('/', 1)[1] if '/' in host_path else '/' | Evidence: print(f'{loc}: {"CONFLICT (robots-blocked)" if blocked else "ok"}')

Reader to Writer Bridge

1. Read: explain spot sitemap urls blocked by robots in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 105.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 107 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {"CONFLICT (robots-blocked)" if blocked else "ok"}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of spot sitemap urls blocked by robots and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Extracting the path after the host and testing it against the disallow prefixes finds sitemap URLs that robots.txt forbids crawling, a contradiction crawlers will flag.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/blog</loc></url>'
       '<url><loc>https://example.com/admin/users</loc></url>'
       '<url><loc>https://example.com/private/notes</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
DISALLOW = ('/private', '/admin')
for u in root.findall('url'):
    loc = u.findtext('loc')
    host_path = loc.split('://', 1)[1]
    path = '/' + host_path.split('/', 1)[1] if '/' in host_path else '/'
    blocked = any(path.startswith(d) for d in DISALLOW)
    print(f'{loc}: {"CONFLICT (robots-blocked)" if blocked else "ok"}')
```

```
output
https://example.com/blog: ok
https://example.com/admin/users: CONFLICT (robots-blocked)
https://example.com/private/notes: CONFLICT (robots-blocked)
```

How it works:

1. DISALLOW holds the prefixes /private and /admin.
2. Each loc is split to recover its path after the host.
3. any(path.startswith(d) for d in DISALLOW) tests the prefixes.
4. The /admin and /private URLs print CONFLICT, /blog prints ok.

Try it:

1. Load the disallow rules from a parsed robots.txt instead of a tuple.
2. Handle wildcard patterns in the disallow rules.
3. Count how many sitemap URLs conflict with robots.

4. Add a `/blog` disallow rule and watch the first URL flip to a conflict.

⚠ Common Mistakes

- A disallowed URL in the sitemap will not be crawled, so listing it wastes the entry.
- Splitting on the first `/` after the scheme can misfire on a bare-domain URL with no path; the guard handles that.
- Robots matching uses prefix rules, so `/admin` also blocks `/administrator`; tighten with a trailing slash if needed.

Skills gained string split | any | robots conflict

→ Where You'll Use This: Cross-checking sitemap paths against robots.txt disallows catches a common mixed-signal mistake where pages are both submitted for indexing and blocked from crawling.

Mental Model

It is mailing an invitation while locking the front gate: the guest has the address but cannot get in, and everyone wastes the trip.

Project 107 Flag Locations With Query Strings

The Idea Behind It

Tracking parameters like utm_source create duplicate URLs that usually should stay out of a sitemap. This project parses the urlset and checks each loc for a question mark, the marker that begins a query string, labelling matches as carrying parameters. The simple membership test surfaces parameterised URLs without needing to dissect the query itself.

What You'll Build & Learn

Flag the sitemap URLs that contain a query string.

- A question mark begins a query string
- Membership test spots parameters
- Tracking parameters duplicate URLs

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 106: spot sitemap urls blocked by robots. Project 108 changes the surface problem while preserving the underlying method.

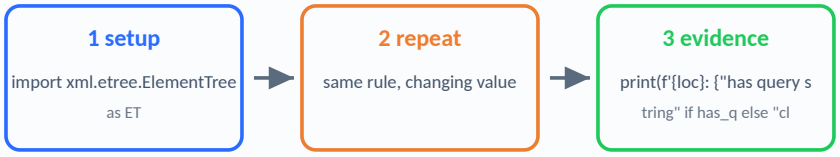
progress check

Reading target: flag locations with query strings is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: {"has query string" if has_q else "clean"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`print(f'{loc}: {"has query string" if has_q else "clean"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: {"has query string" if has_q else "clean"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{loc}: {"has query string" if has_q else "clean"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Locations With Query Strings: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: print(f'{loc}: {"has query string" if has_q else "clean"}') | Evidence: print(f'{loc}: {"has query string" if has_q else "clean"}')

🛠️ Reader to Writer Bridge

1. Read: explain flag locations with query strings in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 106.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 108 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {"has query string" if has_q else "clean"}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag locations with query strings and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A simple `'?' in loc` test detects query strings, which often indicate tracking or faceted variants that dilute the canonical URL and are best excluded from a sitemap.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc></url>'
       '<url><loc>https://example.com/a?utm_source=ad</loc></url>'
       '<url><loc>https://example.com/b?id=5</loc></url>'
       '</urlset>')
root = ET.fromstring(xml)
for u in root.findall('url'):
    loc = u.findtext('loc')
    has_q = '?' in loc
    print(f'{loc}: {"has query string" if has_q else "clean"}')
```

```
output
https://example.com/a: clean
https://example.com/a?utm_source=ad: has query string
https://example.com/b?id=5: has query string
```

How it works:

1. Three locs include a clean URL and two with query strings.
2. '?' in loc tests for a query string.
3. The utm_source and id URLs match.
4. Each loc prints has query string or clean.

Try it:

1. Extract and print the parameter names from each flagged URL.
2. Allow a whitelist of harmless parameters before flagging.
3. Strip the query string to show the clean canonical URL.
4. Strip the query string and report the canonical path instead.

⚠ Common Mistakes

- Not every query string is bad; a `?id=` may identify a genuinely distinct page.
- An escaped `&` in the XML still parses back to `&`, so the query is intact when you read it.

- A `#fragment` is not a query string and is invisible to crawlers anyway.

Skills gained membership test | query string | parameter audit

→ Where You'll Use This: Flagging query strings catches tracking-parameter and faceted URLs that bloat a sitemap with duplicates, helping keep only clean canonical addresses in the file.

Mental Model

The question mark is a price tag still stuck to the gift: harmless to the buyer, but you peel it off before wrapping the canonical version.

Project 108 Audit Optional Tags per URL

The Idea Behind It

Beyond loc, the tags lastmod, changefreq, and priority are all optional, and this project profiles which each URL carries. It iterates a tuple of field names, using find to test presence, and builds parallel lists of present and missing tags per URL. The completeness profile shows at a glance which pages are richly annotated and which carry only a bare loc.

What You'll Build & Learn

Report which optional tags each URL has and which it lacks.

- find tests whether a child exists
- Iterate a tuple of field names
- Split fields into present and missing

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 107: flag locations with query strings. Project 109 changes the surface problem while preserving the underlying method.

progress check

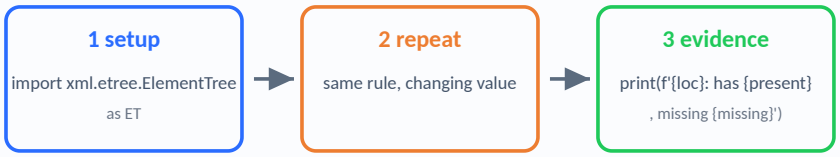
Reading target: audit optional tags per url is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.

- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print(f'{loc}: has {present}, missing {missing}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`present = [f for f in fields if u.find(f) is not None]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: has {present}, missing {missing}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `present = [f for f in fields if u.find(f) is not None]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Audit Optional Tags per URL: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, urlset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: present = [f for f in fields if u.find(f) is not None] | Evidence: print(f'{loc}: has {present}, missing {missing}')

🛠️ Reader to Writer Bridge

1. Read: explain audit optional tags per url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 107.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 109 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: has {present}, missing {missing}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of audit optional tags per url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Iterating a fixed field list and testing `u.find(f) is not None` per URL builds present/missing profiles in a stable, declared order independent of the XML's tag ordering.

```
import xml.etree.ElementTree as ET
xml = ('<urlset>'
       '<url><loc>https://example.com/a</loc><priority>0.8</priority>
<changefreq>daily</changefreq><lastmod>2026-01-01</lastmod></url>'
       '<url><loc>https://example.com/b</loc></url>'
       '<url><loc>https://example.com/c</loc><priority>0.5</priority></url>'
       '</urlset>')
root = ET.fromstring(xml)
fields = ('lastmod', 'changefreq', 'priority')
for u in root.findall('url'):
    loc = u.findtext('loc')
    present = [f for f in fields if u.find(f) is not None]
    missing = [f for f in fields if u.find(f) is None]
    print(f'{loc}: has {present}, missing {missing}')
```

```
output
https://example.com/a: has ['lastmod', 'changefreq', 'priority'], missing []
https://example.com/b: has [], missing ['lastmod', 'changefreq', 'priority']
https://example.com/c: has ['priority'], missing ['lastmod', 'changefreq']
```

How it works:

1. fields lists lastmod, changefreq, and priority.
2. For each url, present keeps fields where find is not None.
3. missing keeps the fields where find returns None.
4. Each loc prints its present and missing lists.

Try it:

1. Score each URL by how many optional tags it carries.
2. Report the share of URLs that have every optional tag.
3. Sort URLs from least to most complete.
4. Add a coverage score as `len(present)` out of three.

⚠ Common Mistakes

- Listing `present` in document order would vary; iterating a fixed `fields` tuple keeps it deterministic.
- Only `<loc>` is required, so a fully bare URL is valid even though it scores zero optional tags.
- `u.find(f)` finds the first child; duplicate optional tags are not detected here.

Skills gained find | list comprehension | completeness profile

→ Where You'll Use This: A completeness profile shows how thoroughly a sitemap uses optional metadata, guiding whether to enrich entries with lastmod and other recrawl hints.

Mental Model

The profile is a checklist on each application form: a quick scan shows which boxes the applicant filled and which they left blank.

Project 109 Catch a Malformed Sitemap

The Idea Behind It

A truncated or mismatched-tag sitemap must fail loudly rather than parse into something wrong. This project feeds ElementTree a broken urlset whose closing tags do not match and wraps the parse in a try/except for ET.ParseError. Catching the error and printing its leading reason turns a crash into a controlled, reportable structural failure.

What You'll Build & Learn

Catch a malformed sitemap's parse error and report it instead of crashing.

- Malformed XML raises ParseError
- try/except contains the failure
- Structural errors should fail loudly

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 108: audit optional tags per url. Project 110 changes the surface problem while preserving the underlying method. *progress check*

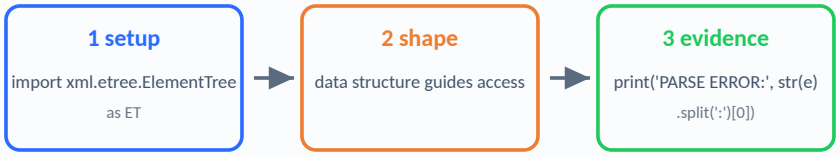
Reading target: catch a malformed sitemap is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The data structure is doing part of the thinking. Ask what the collection stores, how items are found, and what order or uniqueness it promises.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print('PARSE ERROR:', str(e).split(':')[0])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, shape applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`broken = '<urlset><url><loc>https://example.com/a</loc></url><url><loc>https://example.com/b</loc></urlset>'`	This line depends on the chosen data shape. Ask how items are stored, found, or ordered. *slow line*
3. Check the evidence	`print('PARSE ERROR:', str(e).split(':')[0])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
broken = '<urlset><url><loc>https://example.com/a</loc></url><url>
<loc>https://example.com/b</loc></urlset>'
```

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Read the try path as the hoped-for route and the catch/except/finally path as the boundary plan.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Catch a Malformed Sitemap: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, broken. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: broken = '<urlset><url><loc>https://example.com/a</loc></url><url><loc>... | Evidence: print('PARSE ERROR:', str(e).split(':')[0])

🛠️ Reader to Writer Bridge

1. Read: explain catch a malformed sitemap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 108.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 110 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('PARSE ERROR:', str(e).split(':')[0])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of catch a malformed sitemap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The fixture closes a `<url>` with `</urlset>`, so ElementTree raises `ParseError`; catching it turns a crash into a clear diagnostic, which is how a robust audit handles bad input.

```
import xml.etree.ElementTree as ET
broken = '<urlset><url><loc>https://example.com/a</loc></url><url>
<loc>https://example.com/b</loc></urlset>'
try:
    root = ET.fromstring(broken)
    print('parsed', len(root.findall('url')), 'urls')
except ET.ParseError as e:
    print('PARSE ERROR:', str(e).split(':')[0])
```

```
output
PARSE ERROR: mismatched tag
```

How it works:

1. broken closes a url with the urlset tag, a mismatch.
2. The try block calls ET.fromstring on it.
3. ET.ParseError is raised and caught in the except.
4. The handler prints PARSE ERROR with the mismatched tag reason.

Try it:

1. Print the line and column the parser reports for the error.
2. Continue processing a list of sitemaps after one fails.
3. Distinguish a mismatched tag from an unexpected end of file.
4. Fix the closing tag and confirm it parses two URLs.

⚠ Common Mistakes

- A missing closing tag often surfaces as `mismatched tag` rather than `unexpected end`, depending on where it breaks.
- Catching a broad `Exception` would mask unrelated bugs; catch `ET.ParseError` specifically.
- Splitting the message on `:` drops the line/column detail; keep the full string if you need it.

Skills gained try/except | ParseError | error handling

→ Where You'll Use This: Graceful parse-error handling keeps an audit pipeline running over many sitemaps, reporting the broken ones instead of aborting on the first malformed file.

Mental Model

Wrapping the parse is a smoke detector on the file: instead of the whole kitchen burning silently, it sounds an alarm naming what went wrong.

Project 110 Run a Full Sitemap Validation Report

The Idea Behind It

This capstone ties the chapter together by running every earlier check in one pass over a urlset. It parses the file, then per URL validates the https scheme, parses lastmod as a real date, range-checks priority, and tests changefreq against the allowed set, accumulating an error count. A duplicate tally from comparing list length to set size completes the verdict of a basic sitemap linter.

What You'll Build & Learn

Run a combined sitemap linter checking scheme, dates, priority, changefreq, and duplicates.

- Multiple checks combine in one pass
- An error counter aggregates failures
- len versus set size counts duplicates

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 109: catch a malformed sitemap. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

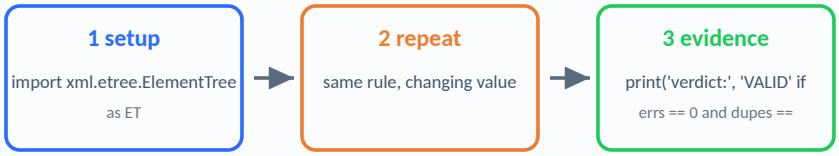
Reading target: run a full sitemap validation report is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import xml.etree.ElementTree as ET` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
- Evidence: `print('verdict:', 'VALID' if errs == 0 and dupes == 0 else 'INVALID')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import xml.etree.ElementTree as ET`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
2. Apply the rule	`if not u.findtext('loc').startswith('https://'):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('verdict:', 'VALID' if errs == 0 and dupes == 0 else 'INVALID')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: XML Sitemaps and URL Discovery becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not u.findtext('loc').startswith('https://'):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run a Full Sitemap Validation Report: connect the surface task to the chapter skill, build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.
Names	Watch these names first: xml, etree, ElementTree, ET, datetime. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import xml.etree.ElementTree as ET | Core: if not u.findtext('loc').startswith('https://'): | Evidence: print('verdict:', 'VALID' if errs == 0 and dupes == 0 else 'INVALID')

🛠️ Reader to Writer Bridge

1. Read: explain run a full sitemap validation report in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 109.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('verdict:', 'VALID' if errs == 0 and dupes == 0 else 'INVALID')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import xml.etree.ElementTree as ET` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run a full sitemap validation report and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

One pass runs every per-field check (scheme, calendar date, priority band, frequency enum) and counts duplicates via set arithmetic, producing the kind of pass/fail summary a sitemap linter emits.

```
import xml.etree.ElementTree as ET
from datetime import datetime
xml = ('<urlset>'
```

```
        '<url><loc>https://example.com/</loc><lastmod>2026-01-01</lastmod>
<priority>1.0</priority><changefreq>daily</changefreq></url>'
        '<url><loc>https://example.com/a</loc><lastmod>2026-13-01</lastmod>
<priority>0.8</priority><changefreq>weekly</changefreq></url>'
        '<url><loc>ftp://example.com/b</loc><lastmod>2026-02-01</lastmod>
<priority>1.5</priority><changefreq>often</changefreq></url>'
        '<url><loc>https://example.com/</loc><lastmod>2026-03-01</lastmod>
<priority>0.5</priority><changefreq>monthly</changefreq></url>'
        '</urlset>')
ALLOWED = {'always', 'hourly', 'daily', 'weekly', 'monthly', 'yearly', 'never'}
root = ET.fromstring(xml)
locs = [u.findtext('loc') for u in root.findall('url')]
errs = 0
for u in root.findall('url'):
    if not u.findtext('loc').startswith('https://'):
        errs += 1
    try:
        datetime.strptime(u.findtext('lastmod'), '%Y-%m-%d')
    except ValueError:
        errs += 1
    if not 0.0 <= float(u.findtext('priority')) <= 1.0:
        errs += 1
    if u.findtext('changefreq') not in ALLOWED:
        errs += 1
dupes = len(locs) - len(set(locs))
print(f'urls={len(locs)} duplicates={dupes} field_errors={errs}')
print('verdict:', 'VALID' if errs == 0 and dupes == 0 else 'INVALID')
```

```
output
urls=4 duplicates=1 field_errors=4
verdict: INVALID
```

How it works:

1. ALLOWED holds the seven changefreq keywords.
2. Per url the code checks the https prefix, parses lastmod, range-checks priority, and tests changefreq, incrementing errs on each failure.
3. dupes is len(locs) minus len(set(locs)).
4. It prints url, duplicate, and error counts, then VALID or INVALID.

Try it:

1. Report which specific check each error came from.
2. Add the absolute-URL and length checks to the pass.
3. Return a structured report dict instead of printing.
4. Print a per-URL breakdown of which check each one failed.

⚠ Common Mistakes

- Counting one error per field per URL can tally several errors on a single bad entry, which is intended here.
- A non-numeric priority would raise inside `float`; a production linter would catch that too.
- Duplicates are detected by exact-string match, so case or slash variants slip past this count.

Skills gained validation pipeline · error counter · set deduplication

→ Where You'll Use This: A single-pass linter combining every rule is the backbone of an automated sitemap-QA step in a deployment pipeline, gating bad files before they go live.

Mental Model

The linter is a pre-flight checklist: one walk around the aircraft confirms fuel, flaps, and tires before the sitemap is cleared for takeoff.

✅ Chapter Progress Checkpoint

What this chapter added: after 40 mini-projects, XML Sitemaps and URL Discovery should feel like a reusable Python move: build and validate urlsets and sitemap indexes — loc, lastmod, priority, size limits and errors.

chapter checkpoint

1 Read
Open any project from Projects 71-110 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 4 (HTTP Status Codes and Redirects) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

8 questions · 4 categories · graded

2 Recall · 2 Meaning · 2 Structure · 2 Transfer

3.1.Q1 Recall · MEDIUM
Turn back to Project 85 (Flag URLs Longer Than 2048 Chars). Without running it, what exactly does it print?

3.1.Q2 Meaning · MEDIUM
Turn back to Project 88 (Find Sitemap URLs Not in the Crawl). Without running it, what exactly does it print?

3.1.Q3 Structure · MEDIUM
Turn back to Project 97 (Measure Sitemap Byte Size). Without running it, what exactly does it print?

3.1.Q4 Transfer MEDIUM Turn back to Project 84 (Escape Ampersands in a URL). Without running it, what exactly does it print?

3.1.Q5 Recall EASY What is the one required child element of every <url> entry in a urlset?

A <lastmod>

B <priority>

C <loc>

D <changefreq>

3.1.Q6 Meaning MEDIUM A single XML sitemap may contain at most how many URLs, and what is the uncompressed size cap?

A 10,000 URLs / 10 MB

B 50,000 URLs / 50 MB

C 100,000 URLs / 100 MB

D No limit / 5 MB

3.1.Q7 Structure EASY You have 130,000 URLs to submit. Describe the file structure that respects the limits.

3.1.Q8 Transfer MEDIUM Why does a URL appearing in your sitemap but never showing up in the crawl deserve investigation?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 3 – XML Sitemaps and URL Discovery

3.1.Q1. `len 2070: TOO LONG` then `len 25: ok`

3.1.Q2. `in sitemap but not crawled: ['https://example.com/orphan']`

3.1.Q3. `sitemap size: 103 bytes (limit 52428800)` then `verdict: ok`

3.1.Q4. `https://example.com/search?q=shoes&sort=price` then `https://example.com/page?a=1&b=2&c=3`

3.1.Q5. C <loc>, the absolute URL. lastmod, priority and changefreq are all optional; a <url> with no <loc> is invalid.

3.1.Q6. B 50,000 URLs and 50 MB uncompressed. Past either limit you split into multiple files referenced by a <sitemapindex>.

3.1.Q7. Split the URLs across at least three child sitemaps (each ⩽ 50,000 URLs and ⩽ 50 MB), then publish one <sitemapindex> whose <sitemap> children each <loc> a child file. The crawler reads the index and fans out to each urlset.

3.1.Q8. The sitemap declared it as a page you want found, yet the crawler never reached it — a sign it may be orphaned (no internal links) or blocked (e.g. by robots.txt). Set-subtracting the crawled set from the sitemap set surfaces these.

Chapter 4 · HTTP Status Codes and Redirects

Classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.

Chapter Skill Map

What this chapter adds: In HTTP Status Codes and Redirects, the focus is classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical. You practise it across 40 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 4 assumes the reading habits from Chapter 3: XML Sitemaps and URL Discovery. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 5, On-Page Signals and the HTML Audit, reuses this chapter's discipline with a new kind of problem.

1 **Projects 111-123**	2 **Projects 124-136**	3 **Projects 137-150**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Every response is a number first, a page second — and the number tells the crawler what to do next.

2xx keep, 3xx follow, 4xx drop, 5xx retry

When a crawler fetches a URL it gets back a status code before it gets any HTML. That three-digit number is the server's verdict on the request, and the *first digit* sorts it into one of four classes: 2xx success (here is the page), 3xx redirect (the page moved, go here instead), 4xx client error (you asked for something that is not here), and 5xx server error (the page should exist but the server is broken right now). Bucketing the code by its leading digit is the very first thing any audit does.

The bucket is just integer division by 100. A handful of lines turns a raw list of codes into the four classes an audit report leads with:

```
codes = [200, 204, 301, 302, 404, 410, 500, 503]
def klass(c):
    return {2: 'success', 3: 'redirect',
            4: 'client-error', 5: 'server-error'}.get(c // 100, 'other')
for c in codes:
    print(c, '->', klass(c))
```

```
output
200 -> success
204 -> success
301 -> redirect
```

```
302 -> redirect
404 -> client-error
410 -> client-error
500 -> server-error
503 -> server-error
```

Why does an SEO care about a number? Because the code decides a page's fate in the index. A `200` page can be indexed and ranked. A `301` hands its hard-won ranking signals to the new URL it points at. A `404` or `410` tells the engine to drop the page. A `5xx` burns crawl budget and, if it persists, gets the page pulled. So reading status codes is not plumbing — it is the difference between a page that ranks and a page that quietly disappears.

the code is the page's fate in the index

The 3xx class is special because the work is not over: a redirect does not contain a page, it contains a forwarding address in its `Location` header. The crawler has to *follow* that address, and the address may itself redirect, and so on. Following each hop until the code is no longer a 3xx is how you resolve a chain to the live page at the end — and that loop is the engine of this entire chapter.

STATUS CODE

The three-digit integer a server returns for every request. Its first digit fixes its class: 2xx success, 3xx redirect, 4xx client error, 5xx server error. Code `200` means OK; the protocol token is written `HTTP/1.1`.

REDIRECT (3XX)

A response that carries no page, only a `Location` header pointing at another URL. The client must fetch that URL next. A `301` is permanent; a `302` is temporary.

301 MOVED PERMANENTLY

Permanent redirect: the resource has a new home for good. Engines pass ranking signals to the target and eventually swap the old URL for the new one in the index.

302 FOUND

Temporary redirect: the resource is briefly elsewhere but its real home stays. Engines keep the original URL indexed, so a 302 used for a permanent move strands link equity.

REDIRECT CHAIN

A sequence of redirects where each hop points at another that also redirects, e.g. A → B → C → live page. Each hop adds latency and bleeds a little link equity; long chains get abandoned.

CANONICAL

A `link rel="canonical"` hint that tells engines which of several live, near-duplicate URLs to index — without removing any of them. A redirect *removes* the duplicate; a canonical *keeps* it.

The four classes, and the codes inside them

A bare number like `301` means little until you attach its reason phrase, so an audit keeps a lookup table from code to meaning. The codes that matter for SEO cluster in each class: `200 OK` (indexable), `301` / `302` (redirects), `404 Not Found` / `410 Gone` (drop it), and `503 Service Unavailable` (a 5xx that says "try again later"). A code not in the table is simply unknown to your report:

```
meanings = {
    200: 'OK', 301: 'Moved Permanently', 302: 'Found',
    304: 'Not Modified', 404: 'Not Found', 410: 'Gone',
    500: 'Internal Server Error', 503: 'Service Unavailable',
}
for code in [200, 301, 404, 503, 418]:
    print(f'{code}: {meanings.get(code, "Unknown")}')
```

```
output
200: OK
301: Moved Permanently
404: Not Found
503: Service Unavailable
418: Unknown
```

The two error codes are not the same instruction. A `404` means "missing, maybe only for now" — the crawler will recrawl it later. A `410` means "gone for good" — the crawler can retire the URL. Returning the right one tells the engine whether to keep knocking or to walk away:

```
pages = [('https://a/', 404), ('https://b/', 410), ('https://c/', 200)]
for url, code in pages:
    if code == 410:
        action = 'retire permanently'
    elif code == 404:
        action = 'recrawl later'
    else:
```

```
        action = 'keep'
    print(f'{code} {url}: {action}')
```

```
output
404 https://a/: recrawl later
410 https://b/: retire permanently
200 https://c/: keep
```

301 versus 302 — permanent or temporary

This is the costliest distinction in the chapter. A `301` says "this moved for good": engines transfer the old URL's ranking signals to the target and replace it in the index. A `302` says "this is a brief detour": engines keep the *original* URL indexed and pass little or nothing to the target. So when you permanently move a page but emit a 302, the link equity never fully transfers — a silent, classic misconfiguration. The fix is to flag any temporary redirect sitting on a permanent move:

```
redirects = [
    {'url': 'https://old-home/', 'status': 302, 'permanent_move': True},
    {'url': 'https://promo/', 'status': 302, 'permanent_move': False},
    {'url': 'https://legacy/', 'status': 301, 'permanent_move': True},
]
for r in redirects:
    misconfigured = r['permanent_move'] and r['status'] == 302
    note = 'should be 301' if misconfigured else 'ok'
    print(f"{r['url']}: {r['status']} ({note})")
```

```
output
https://old-home/: 302 (should be 301)
https://promo/: 302 (ok)
https://legacy/: 301 (ok)
```

The modern protocol also has `308` (permanent, like 301) and `307` (temporary, like 302) which additionally preserve the request method. For SEO the split is the same: `301`/`308` are permanent, `302`/`307` are temporary, and the mix tells you whether a site moves URLs decisively or leaves them in limbo.

Walking a redirect chain to its final URL

A redirect carries a `Location`, and that target may redirect again. To find where a URL really lands you start at the first URL and keep hopping while the status is a 3xx, appending each `Location` to the path, stopping when the code is no longer a redirect. The model below uses a tiny response map keyed by URL; the loop *is* the resolver every redirect audit is built on:

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://b/'},
    'https://b/': {'status': 302, 'location': 'https://c/'},
    'https://c/': {'status': 200, 'location': None},
```

```
}
cur = 'https://a/'
chain = [cur]
while responses[cur]['status'] in (301, 302) and responses[cur]['location']:
    cur = responses[cur]['location']
    chain.append(cur)
print(' -> '.join(chain))
print('final:', cur, 'status', responses[cur]['status'])
```

```
output
https://a/ -> https://b/ -> https://c/
final: https://c/ status 200
```

A healthy chain ends in a `200`. The two failure modes are a chain that dead-ends in a `404` (the target was deleted) or one that lands on a `5xx` (the target exists but is failing) — both send users and crawlers to a wall. And each extra hop costs latency and a sliver of link equity, so audits also *count* the hops and flag any chain longer than a small threshold. *shorter is better; one hop ideally, never a dozen*

Loops, and why you need a "seen" set

If A redirects to B, B to C, and C back to A, a naive resolver loops forever fetching the same URLs. Real browsers and crawlers guard against this two ways: a hard hop limit, and recording every URL already visited in a `seen set` so a repeat means a cycle. The instant the current URL is one you have seen, you stop and flag the loop instead of spinning:

```
responses = {
    'https://x/': {'status': 301, 'location': 'https://y/'},
    'https://y/': {'status': 301, 'location': 'https://z/'},
    'https://z/': {'status': 302, 'location': 'https://x/'},
}
cur = 'https://x/'
seen = set()
loop = False
while responses[cur]['status'] in (301, 302):
    if cur in seen:
        loop = True
        break
    seen.add(cur)
    cur = responses[cur]['location']
print('LOOP at ' + cur if loop else 'ends at ' + cur)
```

```
output
LOOP at https://x/
```

The tightest loop of all is a URL whose `Location` equals itself — an immediate crawl trap you can catch before any walk begins, just by comparing the target to the URL. Whatever the shape, a loop never yields a live page, so the engine indexes nothing.

Canonical versus redirect — keep both, or move one

When two URLs hold the same content you can consolidate them two ways. A `301 redirect` *removes* the duplicate: it sends users and signals to one URL and the other stops serving its own page. A `canonical` *keeps both live* but whispers to engines "index this one." The deciding question is whether users still need the duplicate URL. A tracking-parameter variant like `?ref=ad` must keep working for the people who clicked the ad, so it gets a canonical; a retired product URL nobody needs gets a 301:

```
pages = [
    {'url': 'https://shop/item?ref=ad', 'duplicate_of': 'https://shop/item',
'users_need_it': True},
    {'url': 'https://shop/old-item', 'duplicate_of': 'https://shop/item',
'users_need_it': False},
]
for p in pages:
    tool = 'canonical' if p['users_need_it'] else '301 redirect'
    print(f"{p['url']} -> {tool} to {p['duplicate_of']}")
```

```
output
https://shop/item?ref=ad -> canonical to https://shop/item
https://shop/old-item -> 301 redirect to https://shop/item
```

One last trap on the 200 side: a `soft-404`. The server returns `200 OK` but the body says "Sorry, page not found." The code lies, so the engine wastes budget indexing an empty apology. You catch it by reading the status *and* the body — a 200 whose text admits the page is missing is the tell.

Example 1

A crawl returns these codes: [200, 301, 200, 404, 500, 302, 200, 403, 503, 200]. Summarise the crawl by status class.

SOLUTION

1. Take the first digit of each code with integer division by 100, then attach `xx`: `200` -> `2xx`, `301` -> `3xx`, `404` -> `4xx`, `500` -> `5xx`, and so on.
2. Tally each bucket: the four 200s are `2xx`; the 301 and 302 are `3xx`; the 404 and 403 are `4xx`; the 500 and 503 are `5xx`.
3. Report the four counts — this rolled-up histogram is the headline of any crawl report.

Result: `2xx: 4, 3xx: 2, 4xx: 2, 5xx: 2`. Ten codes compress to four numbers an audit can read at a glance. *first digit is the whole story*

Example 2

A URL resolves through this map: https://old/ returns 301 to https://mid/, which returns 301 to https://new/, which returns 200. Is this a healthy redirect, and how many hops?

SOLUTION

1. Start at `https://old/` and follow `Location` while the status is a 301 or 302: `old` (301) → `mid` (301) → `new`.
2. At `https://new/` the status is `200`, not a 3xx, so the walk stops there — that is the final URL.
3. Count the hops taken: old→mid and mid→new is `2 hops`; the final code is 200, so the chain delivers a live page.

Healthy: it ends at `https://new/` with `200` after 2 hops. A chain that ended in 404 or 503 would fail this same check. *PASS only if the last code is 200*

Example 3

Two pages duplicate https://shop/item: the variant https://shop/item?ref=ad (users still click it from an ad) and the retired https://shop/legacy-item (nobody needs it). Which gets a canonical and which gets a 301?

SOLUTION

1. Ask the deciding question for each: does a user still need this exact URL to keep working?
2. The `?ref=ad` variant must keep serving its page for ad clickers, so removing it with a redirect would break their landing — use a `canonical` pointing at `https://shop/item`.
3. The legacy URL serves no one, so consolidate hard: a `301 redirect` to `https://shop/item` moves users and passes signals.

`?ref=ad` gets a canonical (keep it live); `legacy-item` gets a 301 (remove it). Need-it stays with a canonical, dead-weight gets redirected. *keep what users need, redirect what they do not*

⚡ Mental-Math Shortcut

First digit, then follow. Read the leading digit to learn the verdict — 2 keep, 3 follow, 4 drop, 5 retry — and only the 3xx makes you do more work: follow its `Location` until the code stops starting with 3, carrying a `seen` set so a cycle stops you. A redirect is healthy only when that walk ends on a 200, and a permanent move must answer with 301, never 302.

⚠ Common Mistakes

- Using a 302 (temporary) for a move that is actually permanent — engines keep the old URL indexed and the new page never inherits the old one's ranking signals.
- Letting redirect chains grow (A → B → C → D → page). Each hop adds latency and bleeds equity, and crawlers abandon chains past a few hops, so the final page may never be reached.

- Returning 200 on a "page not found" screen (a soft-404) instead of a real 404 or 410, so the engine wastes budget indexing a page that has no content.
- Pointing a canonical at a URL that itself redirects or 404s — the consolidation hint is wasted, since the named "preferred" URL is not even a live 200 page.

This chapter contains 40 hands-on mini-projects (Projects 111–150). Type each one out, run it, and experiment.

Project 111 Classify a Status Code into Its Class

The Idea Behind It

Integer division by 100 collapses any three-digit status into its leading digit, and a small dictionary maps that digit to a family name. The code holds a list of representative codes, defines klass, and prints each code beside its family. This single bucketing step is where almost every crawl analysis begins because the first digit carries the broad verdict.

What You'll Build & Learn

Sort raw HTTP status numbers into success, redirect, client-error, and server-error families using only the first digit.

- First digit decides the status family
- Integer division as a bucketing trick
- Dictionary get with a fallback default

Skill Roadmap

Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 112 changes the surface problem while preserving the underlying method. *progress check*

Reading target: classify a status code into its class is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `codes = [200, 204, 301, 302, 404, 410, 500, 503]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.

- Evidence: `print(c, '->', klass(c))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`codes = [200, 204, 301, 302, 404, 410, 500, 503]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`for c in codes:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(c, '->', klass(c))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for c in codes:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Classify a Status Code into Its Class: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: codes, klass, success, redirect, client. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: codes = [200, 204, 301, 302, 404, 410, 500, 503] | Core: for c in codes: | Evidence: print(c, '->', klass(c))

🛠️ Reader to Writer Bridge

1. Read: explain classify a status code into its class in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 112 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(c, '->', klass(c))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `codes = [200, 204, 301, 302, 404, 410, 500, 503]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify a status code into its class and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Integer-dividing the code by 100 yields its leading digit, which a small dict maps straight to the human-readable response class.

```
codes = [200, 204, 301, 302, 404, 410, 500, 503]
def klass(c):
    return {2: 'success', 3: 'redirect', 4: 'client-error', 5: 'server-
error'}.get(c // 100, 'other')
for c in codes:
    print(c, '->', klass(c))
```

```
output
200 -> success
204 -> success
301 -> redirect
302 -> redirect
404 -> client-error
410 -> client-error
500 -> server-error
503 -> server-error
```

How it works:

1. A list named codes holds eight sample statuses spanning the 2xx through 5xx range.
2. klass divides the code by 100 and looks the quotient up in a dictionary mapping 2, 3, 4, 5 to family names.
3. When the quotient is missing the get call returns the string other instead of raising.
4. The for loop prints each code, an arrow, and the family that klass returns.

Try it:

1. Add a 1xx informational branch and feed the function a 100 or 101 code.
2. Return a tuple of the digit and the family name rather than just the name.
3. Count how many codes in the list fall into each family using a Counter.
4. Add 199 and confirm it lands in 'other'.

⚠ Common Mistakes

- Comparing the code as a string (`c[0]`) breaks because the codes are integers, not text.
- Using ranges like `200 <= c < 300` per class works but is far more verbose than `c // 100`.
- Forgetting a default for unknown digits raises KeyError on an odd code like 100 or 600.

Skills gained status classification | integer division | dict.get default

→ Where You'll Use This: Every crawler and log analyser groups responses by family first; this digit-based bucketing underpins dashboards that report how many URLs returned success versus error.

Mental Model

Like sorting incoming mail by the first digit of a postcode before anyone reads the street name.

Project 112 Map a Status Code to Its Meaning

The Idea Behind It

A status number alone is opaque to a human reader, so this project keeps a dictionary mapping each code to its standard reason phrase. The loop walks a list of codes and prints each one with its meaning, using get so an unknown code like 418 falls back to the word Unknown rather than crashing. Audit reports need these phrases, not bare integers.

What You'll Build & Learn

Translate numeric status codes into their human-readable reason phrases for a report, defaulting gracefully on unknown codes.

- Reason phrases give codes meaning
- Dictionary lookup with a safe default
- f-string formatting of code and label

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 111: classify a status code into its class. Project 113 changes the surface problem while preserving the underlying method.

progress check

Reading target: map a status code to its meaning is the surface task; the transferable Python move is to mark setup, rule, and output before running.

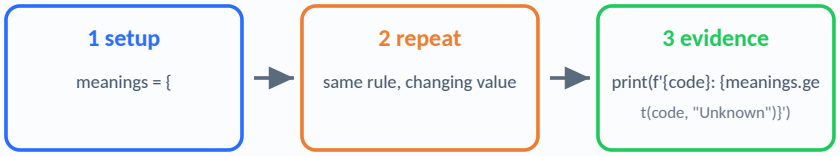
setup -> rule -> result

Logic Lens

- Setup: `meanings = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{code}: {meanings.get(code, "Unknown")}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`meanings = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`for code in [200, 301, 404, 503, 418]:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{code}: {meanings.get(code, "Unknown")}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for code in [200, 301, 404, 503, 418]:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Map a Status Code to Its Meaning: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: meanings, OK, Moved, Permanently, Found. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: meanings = { | Core: for code in [200, 301, 404, 503, 418]: | Evidence: print(f'{code}: {meanings.get(code, "Unknown")}')

🛠️ Reader to Writer Bridge

1. Read: explain map a status code to its meaning in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 111.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 113 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code}: {meanings.get(code, "Unknown")}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `meanings = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of map a status code to its meaning and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A dict of `code -> reason phrase` is the canonical lookup; `.get` with a default keeps an unmapped code (like 418) from crashing the report.

```
meanings = {
    200: 'OK', 301: 'Moved Permanently', 302: 'Found',
    304: 'Not Modified', 404: 'Not Found', 410: 'Gone',
    500: 'Internal Server Error', 503: 'Service Unavailable',
}
```

```
for code in [200, 301, 404, 503, 418]:
    print(f'{code}: {meanings.get(code, "Unknown")}')
```

```
output
200: OK
301: Moved Permanently
404: Not Found
503: Service Unavailable
418: Unknown
```

How it works:

1. The meanings dictionary maps eight common codes to their official reason phrases like OK and Gone.
2. A for loop iterates over a list that deliberately includes 418, which is absent from the dictionary.
3. Each iteration uses meanings.get(code, 'Unknown') so a missing key yields Unknown rather than a KeyError.
4. An f-string prints the code and its resolved phrase separated by a colon.

Try it:

1. Extend the dictionary with 401, 403, and 429 and their reason phrases.
2. Reverse the dictionary so a phrase like Not Found maps back to its code.
3. Print the family from project 111 alongside the reason phrase for each code.
4. Add 307 ('Temporary Redirect') and 308 ('Permanent Redirect').

⚠ Common Mistakes

- Indexing with `meanings[code]` instead of `.get` raises KeyError on the unmapped 418.
- Hard-coding the phrase for 302 as 'Moved Temporarily' uses the old wording; the modern phrase is 'Found'.
- Treating 304 as an error rather than a cache hit misreads its meaning.

Skills gained reason phrases | dict.get default | f-string output

→ Where You'll Use This: Crawl tools and log viewers show reason phrases beside codes so non-engineers can read a report; this lookup is the labelling layer in any status dashboard.

Mental Model

Like a glossary that turns terse error numbers into the plain sentences a reader understands.

Project 113 Walk a Redirect Chain to the Final URL

The Idea Behind It

Redirects are modelled as a dictionary mapping each URL to its status and location. Starting at the first URL, a while loop follows the location of every 301 or 302 hop, appending each new URL to a chain list until it reaches a non-redirecting response. The result is the full path from entry point to the live destination, which is exactly what a browser silently does.

What You'll Build & Learn

Follow a chain of 301 and 302 hops from a starting URL to the final live page and print the whole path.

- A redirect chain is a sequence of hops
- While loops follow location until 200
- Building a path list as you walk

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 112: map a status code to its meaning. Project 114 changes the surface problem while preserving the underlying method.

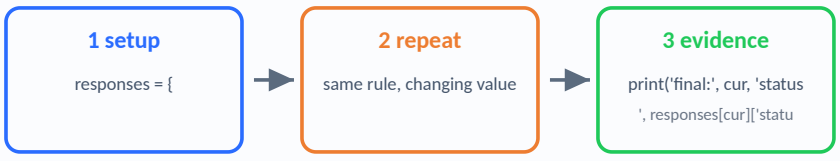
progress check

Reading target: walk a redirect chain to the final url is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('final:', cur, 'status', responses[cur]['status'])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`while responses[cur]['status'] in (301, 302) and responses[cur]['location']:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('final:', cur, 'status', responses[cur]['status'])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `while responses[cur]['status'] in (301, 302) and responses[cur]['location']:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Walk a Redirect Chain to the Final URL: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, cur. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: responses = { | Core: while responses[cur]['status'] in (301, 302) and responses[cur]['locati... | Evidence: print('final:', cur, 'status', responses[cur]['status'])

🛠️ Reader to Writer Bridge

1. Read: explain walk a redirect chain to the final url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 112.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 114 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('final:', cur, 'status', responses[cur]['status'])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of walk a redirect chain to the final url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each hop reads the current URL's response, and while it is a redirect with a `location`, the walk moves to that target until a non-redirect status ends the chain.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://b/'},
    'https://b/': {'status': 302, 'location': 'https://c/'},
    'https://c/': {'status': 200, 'location': None},
}
cur = 'https://a/'
chain = [cur]
while responses[cur]['status'] in (301, 302) and responses[cur]['location']:
    cur = responses[cur]['location']
    chain.append(cur)
print(' -> '.join(chain))
print('final:', cur, 'status', responses[cur]['status'])
```

```
output
https://a/ -> https://b/ -> https://c/
final: https://c/ status 200
```

How it works:

1. The responses dictionary maps three URLs, each to a status code and a location to redirect toward.
2. cur starts at the first URL and chain begins as a list holding that single entry.
3. The while loop continues while the current status is 301 or 302 and a location exists, reassigning cur and appending it.
4. Once cur points at the 200 response the loop stops, and the code prints the joined chain plus the final URL and status.

Try it:

1. Add a fourth hop so the chain runs through five URLs before resolving.
2. Track and print the status code at each hop alongside the URL.
3. Stop the walk early if any hop returns a 404 and report the break.
4. Add a fourth hop and watch the chain grow by one.

⚠ Common Mistakes

- Stopping only on status 200 misses chains that end in a 404 or 500, which are also terminal.
- Appending before checking the next status records a URL that may not exist in the map.
- Treating a 3xx with `location` None as a hop leads to following a missing target.

Skills gained chain following | while loop | list building

→ Where You'll Use This: Resolving redirect chains is core to crawlers and migration checks; you trace where an old URL truly lands before deciding whether the redirect is healthy.

⚡ Mental Model

Like following a trail of forwarding addresses until you reach where someone actually lives now.

Project 114 Catch a Redirect Loop with a Seen Set

💡 The Idea Behind It

When redirects form a cycle, a naive follower spins forever. This code guards against that by keeping a seen set: before following each hop it checks whether the current URL was already visited, and if so it sets a loop flag and breaks. The sample data deliberately points the third URL back to the first, so the walk detects the cycle and reports the URL where it closed.

What You'll Build & Learn

Detect a redirect cycle by recording visited URLs and stopping the walk the moment one repeats.

- A seen set detects revisited URLs
- Cycle detection breaks an infinite loop
- Boolean flag records that a loop occurred

🧭 Skill Roadmap

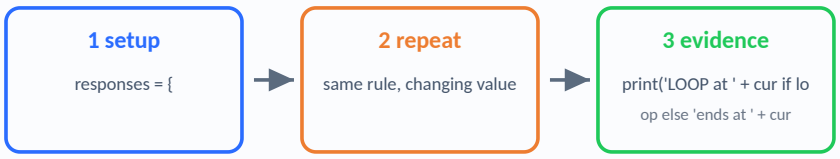
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 113: walk a redirect chain to the final url. Project 115 changes the surface problem while preserving the underlying method. *progress check*

Reading target: catch a redirect loop with a seen set is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('LOOP at ' + cur if loop else 'ends at ' + cur)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus:

Move	Code to inspect	Why it matters
		classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if cur in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('LOOP at ' + cur if loop else 'ends at ' + cur)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if cur in seen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Catch a Redirect Loop with a Seen Set: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, cur. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: responses = { | Core: if cur in seen: | Evidence: print('LOOP at ' + cur if loop else 'ends at ' + cur)

Reader to Writer Bridge

1. Read: explain catch a redirect loop with a seen set in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 113.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 115 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('LOOP at ' + cur if loop else 'ends at ' + cur)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of catch a redirect loop with a seen set and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Adding each URL to `seen` before stepping to its target means re-encountering a seen URL proves a cycle, which the code reports rather than following endlessly.

```
responses = {
    'https://x/': {'status': 301, 'location': 'https://y/'},
    'https://y/': {'status': 301, 'location': 'https://z/'},
    'https://z/': {'status': 302, 'location': 'https://x/'},
}
cur = 'https://x/'
seen = set()
loop = False
while responses[cur]['status'] in (301, 302):
    if cur in seen:
        loop = True
        break
    seen.add(cur)
    cur = responses[cur]['location']
print('LOOP at ' + cur if loop else 'ends at ' + cur)
```

```
output
LOOP at https://x/
```

How it works:

1. The responses dictionary defines three URLs whose locations form a cycle back to the start.
2. cur begins at the first URL, seen is an empty set, and loop starts False.
3. Inside the while loop, if cur is already in seen the code sets loop True and breaks; otherwise it adds cur and advances.
4. A conditional print announces LOOP at the repeated URL or otherwise where the chain ends.

Try it:

1. Print the full visited sequence stored in seen when a loop is found.
2. Convert the response data so no loop exists and confirm the else path runs.
3. Add a maximum hop count as a second safety net alongside the seen set.
4. Point `https://z/` to a fresh URL with status 200 and confirm a clean finish.

⚠ Common Mistakes

- Checking `seen` after advancing misses the very URL that closes the loop.
- Relying on a hop-count cap alone hides whether it is a true loop or just a long chain.
- Clearing `seen` inside the loop defeats the detection entirely.

Skills gained loop detection | set membership | break statement

→ Where You'll Use This: Crawlers must abort redirect loops to avoid hanging; the seen-set pattern protects any chain walker from infinite redirects that waste crawl budget.

Mental Model

Like dropping breadcrumbs through a maze so you know the instant you re-enter a room you already passed.

Project 115 Count the Hops in a Redirect Chain

The Idea Behind It

Rather than recording the whole path, this walker only counts how many redirect hops separate the start from its destination. A hops counter increments on each 301 or 302 step, and the while loop stops at the first non-redirecting response. The chain of four URLs resolves in three hops, turning a sprawling redirect path into one number an audit can compare against a limit.

What You'll Build & Learn

Count the number of redirect hops between a starting URL and its final live destination.

- Hop count summarises a chain length
- Incrementing a counter inside a loop

- Latency and link equity cost per hop

🧭 Skill Roadmap

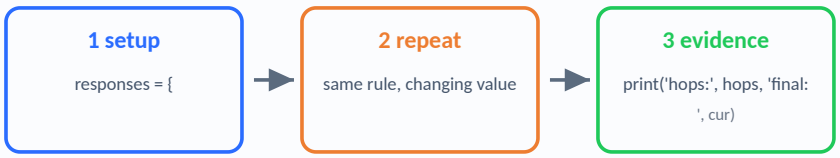
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 114: catch a redirect loop with a seen set. Project 116 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count the hops in a redirect chain is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('hops:', hops, 'final:', cur)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`while responses[cur]['status'] in (301, 302):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print('hops:', hops, 'final:', cur)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `while responses[cur]['status'] in (301, 302):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count the Hops in a Redirect Chain: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, cur. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: while responses[cur]['status'] in (301, 302): | Evidence: print('hops:', hops, 'final:', cur)

Reader to Writer Bridge

1. Read: explain count the hops in a redirect chain in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 114.
   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 116 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('hops:', hops, 'final:', cur)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count the hops in a redirect chain and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Incrementing a counter on every redirect step (not every URL visited) gives the true hop count, which is one less than the number of URLs in the chain.

```
responses = {
    'https://1/': {'status': 301, 'location': 'https://2/'},
    'https://2/': {'status': 301, 'location': 'https://3/'},
    'https://3/': {'status': 302, 'location': 'https://4/'},
    'https://4/': {'status': 200, 'location': None},
}
cur, hops = 'https://1/', 0
while responses[cur]['status'] in (301, 302):
    cur = responses[cur]['location']
    hops += 1
print('hops:', hops, 'final:', cur)
```

```
output
hops: 3 final: https://4/
```

How it works:

1. The responses dictionary defines four URLs forming a three-step redirect chain ending in 200.
2. cur and hops are initialised together, with hops starting at zero.
3. The while loop runs while the current status is 301 or 302, advancing cur and adding one to hops each pass.
4. After the loop the code prints the hop count and the final URL it reached.

Try it:

1. Return both the hop count and the final URL from a function.
2. Raise an error if the hop count exceeds five before resolving.
3. Record the status of each hop in a list as you count.

4. Remove the 302 hop and watch the count drop to 2.

⚠ Common Mistakes

- Counting URLs visited instead of redirects taken overstates the hop count by one.
- Incrementing before moving means the final non-redirect URL is wrongly counted.
- Without a terminal non-redirect entry the `while` would raise KeyError off the end of the map.

Skills gained hop counting | while loop | counter increment

→ Where You'll Use This: SEO audits threshold on hop counts because each extra redirect adds latency and bleeds link equity; this count becomes a per-URL metric in a crawl report.

⚡ Mental Model

Like counting the number of connecting flights before you reach your destination airport.

Project 116 Flag Redirect Chains Longer Than N Hops

💡 The Idea Behind It

Here the chains are pre-resolved into lists of URLs, so the hop count is simply the list length minus one. Iterating the sorted dictionary, each chain is measured and compared against a limit of two; any chain exceeding it is marked FLAG while the rest read ok. This is the reporting step that surfaces the worst redirect offenders from a crawl in one scan.

What You'll Build & Learn

Flag every redirect chain whose hop count exceeds a configured limit, marking offenders for review.

- Hop count is list length minus one
- Comparing a metric against a threshold
- Conditional labelling of offenders

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 115: count the hops in a redirect chain. Project 117 changes the surface problem while preserving the underlying method.

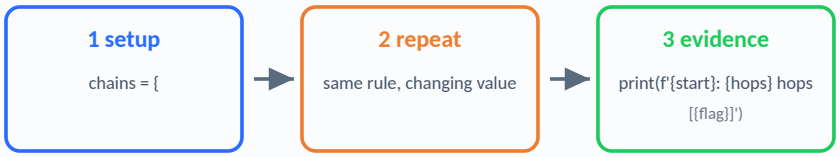
progress check

Reading target: flag redirect chains longer than n hops is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `chains = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{start}: {hops} hops [{flag}]')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`chains = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`flag = 'FLAG' if hops > limit else 'ok'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{start}: {hops} hops [{flag}]')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = 'FLAG' if hops > limit else 'ok'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Redirect Chains Longer Than N Hops: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: chains, https, limit, start, chain. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: chains = { | Core: flag = 'FLAG' if hops > limit else 'ok' | Evidence: print(f'{start}: {hops} hops [{flag}]')

🛠️ Reader to Writer Bridge

1. Read: explain flag redirect chains longer than n hops in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 115.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 117 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{start}: {hops} hops [{flag}]')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `chains = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of flag redirect chains longer than n hops and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each chain's hop count is its length minus one; comparing that to a fixed limit marks chains that exceed the tolerated number of redirects.

```
chains = {
    'https://a/': ['https://a/', 'https://b/'],
    'https://c/': ['https://c/', 'https://d/', 'https://e/', 'https://f/'],
    'https://g/': ['https://g/'],
}
limit = 2
for start, chain in sorted(chains.items()):
    hops = len(chain) - 1
    flag = 'FLAG' if hops > limit else 'ok'
    print(f'{start}: {hops} hops [{flag}]')
```

```
output
https://a/: 1 hops [ok]
https://c/: 3 hops [FLAG]
https://g/: 0 hops [ok]
```

How it works:

1. The chains dictionary maps each starting URL to a pre-resolved list of the URLs in its chain.
2. limit is set to 2 as the maximum tolerated number of hops.
3. Iterating the sorted items, hops is computed as len(chain) minus one for each chain.
4. A conditional sets flag to FLAG when hops exceeds limit and prints the start, count, and flag.

Try it:

1. Make the limit configurable and rerun with a threshold of one.
2. Collect only the flagged chains into a separate list and print it.
3. Sort the output by hop count descending so the worst chain appears first.
4. Lower `limit` to 1 and watch more chains flagged.

⚠ Common Mistakes

- Using `len(chain)` as the hop count over-reports by one because the start URL is included.
- Comparing with `>=` instead of `>` flags chains that sit exactly at the limit.
- Iterating an unsorted dict makes the printed order vary between runs.

Skills gained `threshold flagging` `len-based hop count` `sorted iteration`

→ **Where You'll Use This:** Crawl reports list chains over a tolerance because search engines give up on long chains; this flagging logic decides which redirects an SEO must shorten first.

⚡ Mental Model

Like a quality inspector who pulls any item off the line that exceeds the allowed number of defects.

Project 117 Spot a 302 Where a 301 Belongs

💡 The Idea Behind It

Permanent moves should return 301, but sites frequently serve a temporary 302 by mistake, so link equity never transfers fully. Each redirect record carries a status and a permanent_move boolean, and the code flags any entry that is a permanent move yet returns 302. The output marks the misconfigured one as should be 301 while correct entries read ok.

What You'll Build & Learn

Identify redirects that use a temporary 302 where the move is permanent and a 301 was required.

- 301 versus 302 carries SEO intent
- Combining two conditions with and
- Flagging misconfigured redirects

🧭 Skill Roadmap

`Skill step:` adds one small variation while the chapter pattern is still new. It builds on Project 116: flag redirect chains longer than n hops. Project 118 changes the surface problem while preserving the underlying method.

progress check

`Reading target:` spot a 302 where a 301 belongs is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- `Setup:` `redirects = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- `Writing move:` this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.

- Evidence: `print(f"{r['url']}: {r['status']} ({note})")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`redirects = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`note = 'should be 301' if misconfigured else 'ok'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{r['url']}: {r['status']} ({note})")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `note = 'should be 301' if misconfigured else 'ok'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Spot a 302 Where a 301 Belongs: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: redirects, url, https, old, home. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: redirects = [| Core: note = 'should be 301' if misconfigured else 'ok' | Evidence: print(f"{r['url']}: {r['status']} ({note})")

🛠️ Reader to Writer Bridge

1. Read: explain spot a 302 where a 301 belongs in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 116.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 118 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{r['url']}: {r['status']} ({note})")`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `redirects = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of spot a 302 where a 301 belongs and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A redirect is mislabelled when the move is permanent yet the status is the temporary 302; the audit pairs intent with the code to catch the mismatch.

```
redirects = [
    {'url': 'https://old-home/', 'status': 302, 'permanent_move': True},
    {'url': 'https://promo/', 'status': 302, 'permanent_move': False},
    {'url': 'https://legacy/', 'status': 301, 'permanent_move': True},
]
for r in redirects:
    misconfigured = r['permanent_move'] and r['status'] == 302
    note = 'should be 301' if misconfigured else 'ok'
    print(f"{r['url']}: {r['status']} ({note})")
```

```
output
https://old-home/: 302 (should be 301)
https://promo/: 302 (ok)
https://legacy/: 301 (ok)
```

How it works:

1. A list of redirect dictionaries each holds a url, a status, and a permanent_move boolean.
2. The loop computes misconfigured as True only when permanent_move is set and the status equals 302.
3. note becomes should be 301 for the misconfigured case, otherwise ok.
4. An f-string prints the URL, its status, and the parenthesised note.

Try it:

1. Also flag a 301 used on a clearly temporary, non-permanent move.
2. Add 307 and 308 to the data and decide which are permanent.
3. Count how many redirects in the list are misconfigured overall.
4. Add a 307 temporary redirect and decide whether it counts.

⚠ Common Mistakes

- Flagging every 302 ignores legitimately temporary redirects like a short-lived promo.
- Checking `status != 301` instead of `== 302` would also flag 307/308 you may intend to keep.
- Assuming a missing `permanent_move` key is False can mask a permanent move with no annotation.

Skills gained redirect auditing | boolean conditions | 302 detection

→ Where You'll Use This: A classic technical-SEO audit hunts for permanent moves wrongly served as 302; fixing them to 301 lets accumulated link equity pass to the new URL.

Mental Model

Like noticing a moving company filed a temporary forwarding order when the family actually moved house for good.

Project 118 Detect a Redirect That Dead-Ends in 404

The Idea Behind It

A redirect only helps if it lands on a live page. This walker follows the 301 and 302 hops to the final URL, then inspects that destination's status. The sample chain ends at a URL returning 404, so the code prints the final URL with its status and declares the redirect BROKEN. A chain whose last hop is missing sends users and crawlers straight into a dead end.

What You'll Build & Learn

Follow a redirect chain to its destination and report it broken when that final URL returns 404.

- A chain can dead-end in a 404
- Inspecting the final status after a walk
- Broken redirects waste a crawler's trip

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 117: spot a 302 where a 301 belongs. Project 119 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a redirect that dead-ends in 404 is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.

- Evidence: `print('BROKEN redirect' if final == 404 else 'redirect OK')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`print('BROKEN redirect' if final == 404 else 'redirect OK')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('BROKEN redirect' if final == 404 else 'redirect OK')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('BROKEN redirect' if final == 404 else 'redirect OK')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Redirect That Dead-Ends in 404: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, gone. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: print('BROKEN redirect' if final == 404 else 'redirect OK') | Evidence: print('BROKEN redirect' if final == 404 else 'redirect OK')

🛠️ Reader to Writer Bridge

1. Read: explain detect a redirect that dead-ends in 404 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 117.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 119 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('BROKEN redirect' if final == 404 else 'redirect OK')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a redirect that dead-ends in 404 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Walking to the end of the chain and then inspecting the terminal status reveals a redirect that, despite hopping correctly, resolves to a missing page.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://b/'},
    'https://b/': {'status': 301, 'location': 'https://gone/'},
    'https://gone/': {'status': 404, 'location': None},
}
cur = 'https://a/'
while responses[cur]['status'] in (301, 302):
    cur = responses[cur]['location']
final = responses[cur]['status']
print('final URL:', cur, 'status', final)
print('BROKEN redirect' if final == 404 else 'redirect OK')
```

```
output
final URL: https://gone/ status 404
BROKEN redirect
```

How it works:

1. The responses dictionary chains two redirects into a final URL returning 404.
2. cur starts at the first URL and the while loop advances it while the status is 301 or 302.
3. After the loop, final reads the status of the resolved URL.
4. The code prints the final URL and status, then prints BROKEN redirect when final equals 404.

Try it:

1. Treat any 4xx final status as broken, not only 404.
2. Print the full chain leading up to the broken destination.
3. Suggest a fallback target URL when the final hop is a 404.
4. Make the final status 200 and watch the verdict flip to OK.

⚠ Common Mistakes

- Reporting success as soon as a redirect is followed ignores where it ultimately lands.
- Checking only the first hop's status misses a 404 that appears deeper in the chain.

- Treating a 410 the same as a 404 conflates 'gone forever' with 'not found'.

Skills gained dead-end detection | chain following | final status check

→ Where You'll Use This: Redirect audits verify that migrations point at real pages; a chain ending in 404 means an old URL's traffic and equity drain away to nothing and must be repointed.

⚡ Mental Model

Like following forwarding labels on a parcel only to find the final address is a demolished building.

Project 119 Flag a Redirect Landing on a 5xx Error

💡 The Idea Behind It

A redirect that resolves to a server error is worse than a dead link because the target exists but is failing. The walker follows the hops to the final URL, then checks whether its status sits in the 500-to-599 range. The sample chain ends at a 503, so the output flags SERVER ERROR. Catching a 5xx destination exposes infrastructure problems hiding behind an otherwise valid redirect.

What You'll Build & Learn

Detect when a redirect chain resolves to a 5xx response and flag the failing server destination.

- 5xx means the target is failing
- Range check for the server-error family
- Distinguishing failing from missing targets

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 118: detect a redirect that dead-ends in 404. Project 120 changes the surface problem while preserving the underlying method. *progress check*

Reading target: flag a redirect landing on a 5xx error is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

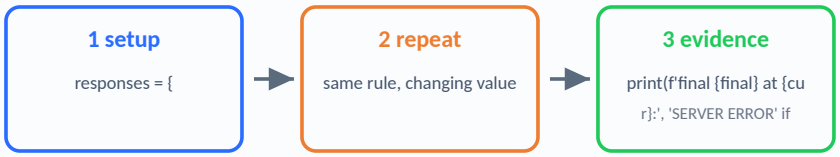
- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from

302 from canonical.

- Evidence: `print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else 'ok')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else 'ok')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else 'ok')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else 'ok')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag a Redirect Landing on a 5xx Error: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, api. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else '... | Evidence: print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else '...

🛠️ Reader to Writer Bridge

1. Read: explain flag a redirect landing on a 5xx error in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 118.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 120 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else 'ok')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag a redirect landing on a 5xx error and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

After resolving the chain, the terminal status is tested against the 500-599 range; a hit means the redirect points at a server that is currently failing.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://api/'},
    'https://api/': {'status': 503, 'location': None},
}
cur = 'https://a/'
while responses[cur]['status'] in (301, 302):
    cur = responses[cur]['location']
final = responses[cur]['status']
server_error = 500 <= final < 600
print(f'final {final} at {cur}:', 'SERVER ERROR' if server_error else 'ok')
```

```
output
final 503 at https://api/: SERVER ERROR
```

How it works:

1. The responses dictionary redirects one URL to an API endpoint that returns 503.
2. The while loop follows 301 and 302 hops until cur reaches the failing endpoint.
3. server_error is True when final is at least 500 and below 600.
4. An f-string prints the final status and URL, appending SERVER ERROR or ok.

Try it:

1. Separate a 503 (temporary) from a 500 in the message.
2. Retry the final fetch a configurable number of times before flagging.
3. Combine this with project 118 to report 4xx and 5xx endings together.
4. Change the final status to 502 and confirm it still flags.

⚠ Common Mistakes

- Hard-coding `final == 500` misses 502, 503 and other 5xx codes.
- Stopping the walk on the first 5xx instead of following all redirects first reports the wrong URL.
- Treating a transient 503 as permanently broken may retire a URL that recovers.

Skills gained `5xx detection` `range comparison` `chain following`

→ **Where You'll Use This:** Monitoring catches redirects pointing at flaky backends; flagging a 5xx destination surfaces an outage behind a redirect that a status-only crawl might miss.

⚡ Mental Model

Like following directions to a shop that is open on paper but has its lights off and doors jammed.

Project 120 Decide Between a Redirect and a Canonical

💡 The Idea Behind It

Duplicate pages can be consolidated two ways, and the right choice depends on whether users still need the duplicate URL. This code reads a users_need_it boolean per page: when users need it, a canonical tag keeps both URLs live; when they do not, a 301 redirect moves them off. The output shows the parameterised URL getting a canonical and the old URL getting a redirect.

What You'll Build & Learn

Choose between a canonical tag and a 301 redirect for each duplicate page based on whether users still need it.

- Canonical keeps a URL live, redirect moves it
- Choice hinges on user need
- Boolean-driven tool selection

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 119: flag a redirect landing on a 5xx error. Project 121 changes the surface problem while preserving the underlying method. *progress check*

Reading target: decide between a redirect and a canonical is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `pages = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.

- Evidence: `print(f"{p['url']} -> {tool} to {p['duplicate_of']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`tool = 'canonical' if p['users_need_it'] else '301 redirect'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{p['url']} -> {tool} to {p['duplicate_of']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `tool = 'canonical' if p['users_need_it'] else '301 redirect'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Decide Between a Redirect and a Canonical: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: pages, url, https, shop, item. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = [| Core: tool = 'canonical' if p['users_need_it'] else '301 redirect' | Evidence: print(f"{p['url']} -> {tool} to {p['duplicate_of']}")

🛠️ Reader to Writer Bridge

1. Read: explain decide between a redirect and a canonical in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 119.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 121 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{p['url']} -> {tool} to {p['duplicate_of']}")`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `pages = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of decide between a redirect and a canonical and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

When the duplicate URL must stay reachable (a tracked link) a canonical is right; when it is obsolete a 301 redirect both consolidates signals and removes the old URL.

```
pages = [
    {'url': 'https://shop/item?ref=ad', 'duplicate_of': 'https://shop/item',
'users_need_it': True},
    {'url': 'https://shop/old-item', 'duplicate_of': 'https://shop/item',
'users_need_it': False},
]
for p in pages:
    tool = 'canonical' if p['users_need_it'] else '301 redirect'
    print(f"{p['url']} -> {tool} to {p['duplicate_of']}")
```

```
output
https://shop/item?ref=ad -> canonical to https://shop/item
https://shop/old-item -> 301 redirect to https://shop/item
```

How it works:

1. A list of page dictionaries each carries a url, its duplicate_of target, and a users_need_it boolean.
2. The loop selects canonical when users_need_it is True and 301 redirect otherwise.
3. An f-string prints the page URL, the chosen tool, and the consolidation target.
4. The result keeps the ad-referral URL via canonical and retires the old item via redirect.

Try it:

1. Add a self-referencing canonical case where the page is its own master.
2. Reject a redirect when the duplicate target is the page itself.
3. Include a reason string explaining each tool choice in the output.
4. Add a page that is not a duplicate and skip both tools.

⚠ Common Mistakes

- Always 301-redirecting breaks URLs that users or campaigns still depend on.
- Always using a canonical leaves dead URLs crawlable instead of retiring them.
- Forgetting that a canonical is a hint while a 301 is enforced overstates the canonical's power.

Skills gained canonical vs redirect | boolean branching | duplicate handling

→ Where You'll Use This: Deciding between canonicals and redirects is a daily SEO judgement; parameter URLs that users hit get canonicals, while genuinely retired URLs get 301s to consolidate signals.

⚡ Mental Model

Like deciding whether to forward someone's mail entirely or just note on file which address is their main one.

Project 121 Detect a Soft-404 Behind a 200

💡 The Idea Behind It

A soft-404 returns status 200 but its body admits the page is missing, so engines waste budget indexing nothing. This code holds a tuple of marker phrases and checks each page: it is a soft-404 when the status is 200 and any marker appears in the lowercased body. The page whose body says page not found is caught, while the genuine 404 with the same words is not, since its status already tells the truth.

What You'll Build & Learn

Catch soft-404 pages that return status 200 yet carry not-found language in their body text.

- Soft-404 mismatches status and body
- any with a generator over markers
- Lowercasing body before matching

🧭 Skill Roadmap

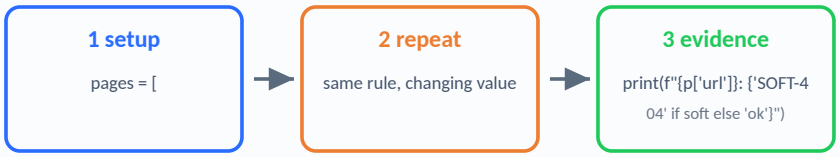
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 120: decide between a redirect and a canonical. Project 122 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect a soft-404 behind a 200 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `pages = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}")` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}")`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Soft-404 Behind a 200: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: pages, url, https, status, body. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = [| Core: print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}") | Evidence: print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}")

🛠️ Reader to Writer Bridge

1. Read: explain detect a soft-404 behind a 200 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 120.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 122 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}")`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `pages = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a soft-404 behind a 200 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A soft-404 is the contradiction of a 200 status with body text signalling absence; lower-casing the body and scanning for marker phrases reveals it.

```
pages = [
    {'url': 'https://a/', 'status': 200, 'body': 'Welcome to our store'},
    {'url': 'https://b/', 'status': 200, 'body': 'Sorry, page not found'},
    {'url': 'https://c/', 'status': 404, 'body': 'Not Found'},
]
markers = ('not found', 'no longer exists', 'does not exist')
```

```
for p in pages:
    soft = p['status'] == 200 and any(m in p['body'].lower() for m in markers)
    print(f"{p['url']}: {'SOFT-404' if soft else 'ok'}")
```

```
output
https://a/: ok
https://b/: SOFT-404
https://c/: ok
```

How it works:

1. A list of page dictionaries holds a url, a status, and a body string.
2. markers is a tuple of not-found phrases to search for inside the body.
3. soft is True when the status is 200 and any marker appears in the lowercased body via any.
4. The loop prints SOFT-404 for the offending 200 page and ok for the rest.

Try it:

1. Add more marker phrases such as page removed and content unavailable.
2. Report which specific marker matched for each soft-404.
3. Recommend returning a real 404 or 410 for each flagged page.
4. Add a real product page containing the word 'found' in another context and check for false positives.

⚠ Common Mistakes

- Trusting the status alone never catches a soft-404, since it really is 200.
- Case-sensitive matching misses 'Not Found' when you only search for 'not found'.
- Flagging the genuine 404 as a soft-404 confuses an honest error with a disguised one.

Skills gained soft-404 detection | substring matching | any() over markers

→ Where You'll Use This: Crawlers and SEO tools flag soft-404s because they bloat the index with empty pages; comparing the served status against body language is the standard way to expose them.

Mental Model

Like a store hanging an Open sign on a door while a note inside reads Sorry, we have closed.

Project 122 Count the 4xx Errors in a Crawl

The Idea Behind It

After a crawl, the share of client errors shows how many links point at missing or forbidden pages. This code holds crawl results as tuples of URL and status, then uses a list comprehension to gather every URL whose

status falls in the 400-to-499 range. It prints the count and the sorted offending URLs. Counting 4xx responses is the first health metric an audit reads.

What You'll Build & Learn

Count the 4xx client-error responses in a crawl and list the offending URLs.

- 4xx responses signal broken targets
- List comprehension with a range filter
- Sorting URLs for a readable report

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 121: detect a soft-404 behind a 200. Project 123 changes the surface problem while preserving the underlying method.

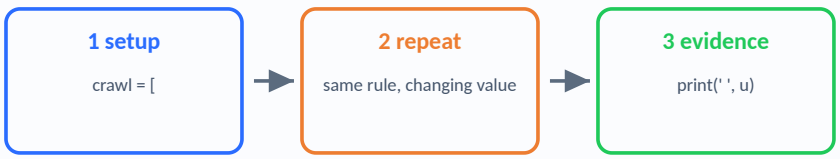
progress check

Reading target: count the 4xx errors in a crawl is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `crawl = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(' ', u)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`crawl = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`fourxx = [u for u, s in crawl if 400 <= s < 500]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' ', u)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `fourxx = [u for u, s in crawl if 400 <= s < 500]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count the 4xx Errors in a Crawl: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: crawl, https, fourxx, xx, count. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: crawl = [| Core: fourxx = [u for u, s in crawl if 400 <= s < 500] | Evidence: print(' ', u)

Reader to Writer Bridge

1. Read: explain count the 4xx errors in a crawl in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 121.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 123 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' ', u)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `crawl = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count the 4xx errors in a crawl and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering the crawl for statuses in the 400-499 range isolates every client error; the length is the count and the URLs name the offenders.

```
crawl = [
    ('https://a/', 200), ('https://b/', 404),
    ('https://c/', 301), ('https://d/', 403),
    ('https://e/', 200), ('https://f/', 410),
]
fourxx = [u for u, s in crawl if 400 <= s < 500]
print('4xx count:', len(fourxx))
for u in sorted(fourxx):
    print(' ', u)
```

```
output
4xx count: 3
  https://b/
  https://d/
  https://f/
```

How it works:

1. crawl is a list of tuples pairing each URL with its returned status code.
2. A comprehension keeps each URL whose status is at least 400 and below 500.
3. The code prints the length of that list as the 4xx count.
4. It then loops over the sorted URLs, printing each indented beneath the count.

Try it:

1. Break the count down by specific code, separating 404 from 403.
2. Express the 4xx count as a percentage of the whole crawl.
3. Also list which source pages link to each 4xx target.
4. Also tally the 5xx errors separately.

⚠ Common Mistakes

- Testing `s == 404` alone misses 403, 410 and other client errors.
- Using `s > 400` excludes 400 itself and includes 5xx codes.
- Counting redirects (3xx) as errors inflates the figure.

Skills gained 4xx counting | list comprehension | range filter

→ Where You'll Use This: The client-error count headlines crawl health reports; a rising 4xx tally tells an SEO that broken links or removed pages are accumulating across the site.

Mental Model

Like tallying how many addresses on a delivery route turned out to be wrong or off-limits.

Project 123 Compute the 5xx Error Rate of a Crawl

The Idea Behind It

Server errors during a crawl can shrink crawl budget and drop pages from the index, so this code expresses 5xx as a rate. It counts statuses in the 500-to-599 range with a generator inside sum, divides by the total, and rounds the percentage to one decimal. The sample gives three of eight as 37.5 percent. A rate makes spikes comparable across crawls of different sizes.

What You'll Build & Learn

Compute the percentage of server-error responses in a crawl so spikes compare across crawl sizes.

- 5xx rate as a comparable metric
- Counting with sum and a generator
- Rounding a percentage to one decimal

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 122: count the 4xx errors in a crawl. Project 124 changes the surface problem while preserving the underlying method.

progress check

Reading target: compute the 5xx error rate of a crawl is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `statuses = [200, 200, 500, 200, 503, 301, 200, 502]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{server_errors}/{total} server errors = {rate}%')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`statuses = [200, 200, 500, 200, 503, 301, 200, 502]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`server_errors = sum(1 for s in statuses if 500 <= s < 600)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{server_errors}/{total} server errors = {rate}%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `server_errors = sum(1 for s in statuses if 500 <= s < 600)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute the 5xx Error Rate of a Crawl: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: statuses, total, len, server_errors, sum. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: statuses = [200, 200, 500, 200, 503, 301, 200, 502] | Core: server_errors = sum(1 for s in statuses if 500 <= s < 600) | Evidence: print(f'{server_errors}/{total} server errors = {rate}%')

🛠️ Reader to Writer Bridge

1. Read: explain compute the 5xx error rate of a crawl in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 122.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 124 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{server_errors}/{total} server errors = {rate}%')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `statuses = [200, 200, 500, 200, 503, 301, 200, 502]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute the 5xx error rate of a crawl and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Counting statuses in the 500-599 band and dividing by the total, then scaling to a percentage and rounding, yields a stable error-rate figure.

```
statuses = [200, 200, 500, 200, 503, 301, 200, 502]
total = len(statuses)
server_errors = sum(1 for s in statuses if 500 <= s < 600)
rate = round(100 * server_errors / total, 1)
print(f'{server_errors}/{total} server errors = {rate}%')
```

```
output
3/8 server errors = 37.5%
```

How it works:

1. statuses is a list of eight sample status codes including three server errors.
2. total captures the length of the list.
3. server_errors sums one for each status in the 500-to-599 range using a generator inside sum.
4. rate is the rounded percentage, and an f-string prints the fraction and percent.

Try it:

1. Compute a 4xx rate alongside the 5xx rate in the same report.
2. Trigger an alert string when the rate crosses a configurable threshold.
3. Round to two decimals and compare the output difference.
4. Add more 200s and watch the rate fall.

⚠ Common Mistakes

- Integer division (`server_errors // total`) collapses small rates to 0.
- Forgetting `round` prints a long noisy float that varies in appearance.
- Including 3xx or 4xx in the numerator measures the wrong kind of failure.

Skills gained error rate | sum generator | percentage rounding

→ Where You'll Use This: Monitoring tracks the 5xx rate over time; a spike warns that the server is faltering under crawl load, which can cost crawl budget and de-index pages.

Mental Model

Like reporting failures per hundred attempts rather than a raw count, so two factories of different sizes compare fairly.

Project 124 Verify a Redirect Chain Ends in 200

The Idea Behind It

A healthy redirect ultimately delivers a live page, and this is the pass-or-fail check every audit runs. The walker follows the 301 and 302 hops to the destination, then tests whether that final status equals 200. The sample chain of two permanent redirects ends at a 200, so the code prints PASS with the resolved URL and status. Anything else would print FAIL.

What You'll Build & Learn

Verify that a redirect chain resolves to a 200 response and report a clear pass or fail.

- A healthy chain ends in 200
- Equality test on the final status
- Pass-fail reporting for audits

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 123: compute the 5xx error rate of a crawl. Project 125 changes the surface problem while preserving the underlying method.

progress check

Reading target: verify a redirect chain ends in 200 is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

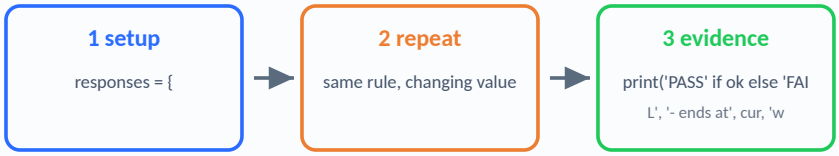
- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('PASS' if ok else 'FAIL', '– ends at', cur, 'with', responses[cur]['status'])` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`print('PASS' if ok else 'FAIL', '- ends at', cur, 'with', responses[cur]['status'])`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('PASS' if ok else 'FAIL', '- ends at', cur, 'with', responses[cur]['status'])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('PASS' if ok else 'FAIL', '- ends at', cur, 'with', responses[cur]['status'])`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Verify a Redirect Chain Ends in 200: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, old, status, location. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: print('PASS' if ok else 'FAIL', '- ends at', cur, 'with', responses[cur... | Evidence: print('PASS' if ok else 'FAIL', '- ends at', cur, 'with', responses[cur...

🛠️ Reader to Writer Bridge

1. Read: explain verify a redirect chain ends in 200 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 123.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 125 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('PASS' if ok else 'FAIL', '- ends at', cur, 'with', responses[cur]['status'])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of verify a redirect chain ends in 200 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Resolving the chain to its terminal URL and asserting the status equals 200 separates redirects that deliver content from those that fail.

```
responses = {
    'https://old/': {'status': 301, 'location': 'https://mid/'},
    'https://mid/': {'status': 301, 'location': 'https://new/'},
    'https://new/': {'status': 200, 'location': None},
}
```

```
cur = 'https://old/'
while responses[cur]['status'] in (301, 302):
    cur = responses[cur]['location']
ok = responses[cur]['status'] == 200
print('PASS' if ok else 'FAIL', '- ends at', cur, 'with', responses[cur]
['status'])
```

```
output
PASS - ends at https://new/ with 200
```

How it works:

1. The responses dictionary chains two 301 redirects into a final URL returning 200.
2. cur starts at the old URL and the while loop advances through the 301 hops.
3. ok is True only when the resolved status equals exactly 200.
4. The code prints PASS or FAIL along with the final URL and its status.

Try it:

1. Accept any 2xx status, not strictly 200, as a pass.
2. Run the check over a dictionary of several chains and tally passes.
3. Report the hop count alongside the pass-fail verdict.
4. Change the final status to 404 and watch it FAIL.

⚠ Common Mistakes

- Asserting 200 on the first hop, not the last, passes a chain that later breaks.
- Accepting any non-3xx status as success would pass a chain ending in 404.
- Missing the terminal map entry raises KeyError when the loop reads it.

Skills gained pass-fail check | chain following | equality test

→ Where You'll Use This: Post-migration verification runs this exact test across the redirect map, confirming each old URL now lands on a real 200 page before declaring the migration successful.

⚡ Mental Model

Like confirming a forwarded letter actually arrived at a real mailbox rather than vanishing en route.

Project 125 Normalise Trailing-Slash Redirects

💡 The Idea Behind It

Many servers enforce one canonical form by redirecting /path to /path/, and this code models that trailing-slash rule. canonicalise returns the URL unchanged if it already ends with a slash, otherwise it appends one.

The loop prints a 200 for URLs that are already canonical and a 301 to the slashed target for those that need rewriting, showing exactly which URLs move and where.

What You'll Build & Learn

Model a trailing-slash canonicalisation rule, issuing a 301 to the slashed form when needed.

- Trailing slash defines one canonical form
- endswith guards the transform
- Already-canonical URLs return 200

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 124: verify a redirect chain ends in 200. Project 126 changes the surface problem while preserving the underlying method.

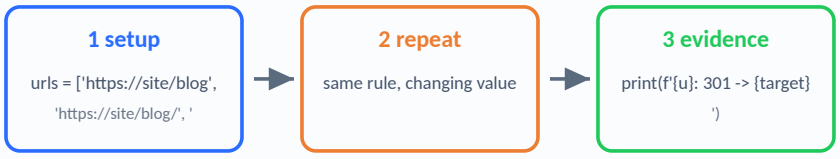
progress check

Reading target: normalise trailing-slash redirects is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `urls = ['https://site/blog', 'https://site/blog/', 'https://site/about']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{u}: 301 -> {target}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = ['https://site/blog', 'https://site/blog/', 'https://site/about']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`return u if u.endswith('/') else u + '/'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{u}: 301 -> {target}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return u if u.endswith('/') else u + '/'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Normalise Trailing-Slash Redirects: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: urls, https, site, blog, about. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: urls = ['https://site/blog', 'https://site/blog/', 'https://site/about'] | Core: return u if u.endswith('/') else u + '/' | Evidence: print(f'{u}: 301 -> {target}')

🛠️ Reader to Writer Bridge

1. Read: explain normalise trailing-slash redirects in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 124.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 126 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{u}: 301 -> {target}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `urls = ['https://site/blog', 'https://site/blog/', 'https://site/about']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of normalise trailing-slash redirects and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The policy here is 'always end in a slash'; a URL already ending in `/` returns 200, while one without gets a 301 to its slashed form.

```
urls = ['https://site/blog', 'https://site/blog/', 'https://site/about']
def canonicalise(u):
    return u if u.endswith('/') else u + '/'
for u in urls:
    target = canonicalise(u)
    if target == u:
        print(f'{u}: 200 (already canonical)')
    else:
        print(f'{u}: 301 -> {target}')
```

```
output
https://site/blog: 301 -> https://site/blog/
https://site/blog/: 200 (already canonical)
https://site/about: 301 -> https://site/about/
```

How it works:

1. urls is a list with one already-slashed URL and two without a trailing slash.

2. canonicalise returns the URL as-is when it ends with a slash, else appends one.
3. For each URL the code computes target and compares it to the original.
4. When target equals the URL it prints 200 already canonical, otherwise a 301 to the target.

Try it:

1. Reverse the rule to strip trailing slashes instead of adding them.
2. Leave file-like paths ending in a dot extension unchanged.
3. Skip the homepage so the bare domain is not double-slashed.
4. Switch the policy to strip trailing slashes instead.

⚠ Common Mistakes

- Appending a slash to a URL that already has one creates a double slash.
- Applying the rule to the root URL when the policy differs there can cause a loop.
- Picking the opposite policy (strip the slash) silently reverses every verdict.

Skills gained `trailing slash` `endswith check` `301 modelling`

→ **Where You'll Use This:** Servers normalise slash variants to prevent duplicate URLs; modelling the rule shows an SEO which paths a 301 will rewrite and confirms the canonical form is consistent.

Mental Model

Like a style guide insisting every entry ends with a period, fixing the ones that forgot it.

Project 126 Enforce a www-to-Non-www Redirect

The Idea Behind It

A site should pick one host, with or without www, and 301 the other so ranking signals do not split. to_nonwww replaces the https://www. prefix with https:// exactly once, leaving already-bare hosts untouched. The loop prints a 200 for URLs already on the non-www host and a 301 to the rewritten target for www URLs, encoding a single non-www canonical policy.

What You'll Build & Learn

Enforce a non-www canonical host, issuing a 301 from any www URL to its bare-host equivalent.

- One host avoids split signals
- replace with a count limit of one
- Non-www as the canonical policy

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 125: normalise trailing-slash redirects. Project 127 changes the surface problem while preserving the underlying method.

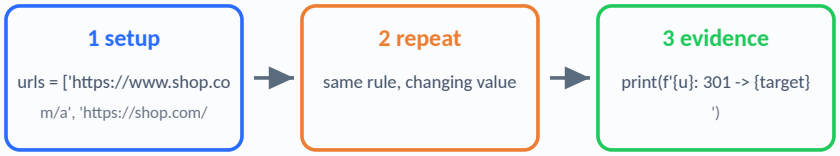
progress check

Reading target: enforce a www-to-non-www redirect is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `urls = ['https://www.shop.com/a', 'https://shop.com/b', 'https://www.shop.com/']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{u}: 301 -> {target}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = ['https://www.shop.com/a', 'https://shop.com/b', 'https://www.shop.com/']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if target == u:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{u}: 301 -> {target}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if target == u:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Enforce a www-to-Non-www Redirect: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: urls, https, www, shop, com. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: urls = ['https://www.shop.com/a', 'https://shop.com/b', 'https://www.sh... | Core: if target == u: | Evidence: print(f'{u}: 301 -> {target}')

Reader to Writer Bridge

1. Read: explain enforce a www-to-non-www redirect in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 125.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 127 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{u}: 301 -> {target}')`. Then compare punctuation, spacing, and order.
2. Explain: point to `urls = ['https://www.shop.com/a', 'https://shop.com/b', 'https://www.shop.com/']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of enforce a www-to-non-www redirect and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Stripping a leading `www.` from the host (only the first occurrence) maps every `www` URL to its canonical non-www twin; URLs already canonical return 200.

```
urls = ['https://www.shop.com/a', 'https://shop.com/b',
'https://www.shop.com/']
def to_nonwww(u):
    return u.replace('https://www.', 'https://', 1)
for u in urls:
    target = to_nonwww(u)
    if target == u:
        print(f'{u}: 200')
    else:
        print(f'{u}: 301 -> {target}')
```

```
output
https://www.shop.com/a: 301 -> https://shop.com/a
https://shop.com/b: 200
https://www.shop.com/: 301 -> https://shop.com/
```

How it works:

1. urls mixes two www-prefixed URLs with one already on the bare host.
2. to_nonwww replaces the https://www. prefix with https:// using a count of one.
3. Each URL's target is compared to the original to detect a change.
4. Matching URLs print 200, while www URLs print a 301 to the non-www target.

Try it:

1. Invert the policy to force www and redirect bare hosts to it.
2. Handle http URLs by also upgrading them to https.
3. Only rewrite the host when it begins exactly with www, not any subdomain.
4. Flip the policy to redirect non-www to www.

⚠ Common Mistakes

- Replacing `www.` anywhere in the URL could mangle a path segment that contains 'www.'.
- Omitting the count argument to `replace` risks editing more than the host.
- Choosing non-www here but linking internally to www elsewhere reopens the split.

Skills gained www canonicalisation | string replace | host policy

→ Where You'll Use This: Choosing www or non-www and redirecting the other is foundational SEO hygiene; this models the rule that keeps link equity and rankings on a single host.

⚡ Mental Model

Like a company deciding all email goes to one official address and bouncing anything sent to the old alias.

Project 127 Redirect Mixed-Case URLs to Lowercase

💡 The Idea Behind It

URL paths are case-sensitive, so /Page and /page split signals between two URLs. lower_path splits the scheme from the rest, partitions the host from the path, and lowercases only the path while leaving the host intact. The loop prints a 200 for already-lowercase URLs and a 301 to the lowercased target otherwise, enforcing a single canonical casing without touching the domain.

What You'll Build & Learn

Redirect mixed-case URL paths to a lowercase canonical form while preserving the host casing.

- Paths are case-sensitive, hosts are not
- split and partition isolate the path
- Lowercasing only the path component

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 126: enforce a www-to-non-www redirect. Project 128 changes the surface problem while preserving the underlying method.

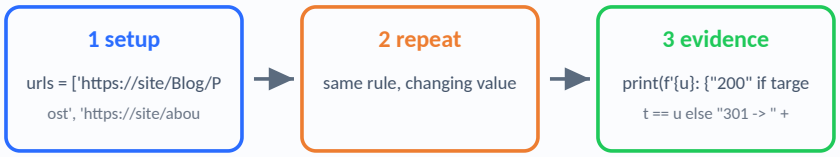
progress check

Reading target: redirect mixed-case urls to lowercase is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `urls = ['https://site/Blog/Post', 'https://site/about', 'https://site/FAQ']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{u}: {"200" if target == u else "301 -> " + target}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = ['https://site/Blog/Post', 'https://site/about', 'https://site/FAQ']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`print(f'{u}: {"200" if target == u else "301 -> " + target}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{u}: {"200" if target == u else "301 -> " + target}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{u}: {"200" if target == u else "301 -> " + target}')`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Redirect Mixed-Case URLs to Lowercase: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: urls, https, site, Blog, Post. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: urls = ['https://site/Blog/Post', 'https://site/about', 'https://site/F... | Core: print(f'{u}: {"200" if target == u else "301 -> " + target}') | Evidence: print(f'{u}: {"200" if target == u else "301 -> " + target}')

🛠️ Reader to Writer Bridge

1. Read: explain redirect mixed-case urls to lowercase in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 126.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 128 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{u}: {"200" if target == u else "301 -> " + target}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `urls = ['https://site/Blog/Post', 'https://site/about', 'https://site/FAQ']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of redirect mixed-case urls to lowercase and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Only the path is lowercased while the scheme and host are preserved, since hosts are case-insensitive but lowercasing the whole URL would also corrupt query values.

```
urls = ['https://site/Blog/Post', 'https://site/about', 'https://site/FAQ']
def lower_path(u):
    scheme, rest = u.split('://', 1)
    host, sep, path = rest.partition('/')
    return scheme + '://' + host + sep + path.lower()
for u in urls:
    target = lower_path(u)
    print(f'{u}: {"200" if target == u else "301 -> " + target}')
```

```
output
https://site/Blog/Post: 301 -> https://site/blog/post
https://site/about: 200
https://site/FAQ: 301 -> https://site/faq
```

How it works:

1. urls holds one mixed-case path, one already lowercase, and one uppercase segment.
2. lower_path splits on :// to separate scheme, then partitions the rest into host, separator, and path.
3. It rebuilds the URL with the host unchanged and the path lowercased.
4. Each URL prints 200 when already lowercase or a 301 to the lowercased target.

Try it:

1. Preserve case inside query strings while lowercasing only the path.
2. Also collapse repeated slashes in the path during the rewrite.
3. Leave case-sensitive token segments, such as base64 ids, untouched.
4. Add a URL with an uppercase query value and confirm the path-only fold.

⚠ Common Mistakes

- Lowercasing the entire URL would also fold case-sensitive query parameters.
- Splitting on `/` without `partition` raises when a URL has no path.

- Lowercasing the host is harmless but lowercasing the path of a case-sensitive backend can 404.

Skills gained case canonicalisation | string partition | path lowercasing

→ Where You'll Use This: Mixed-case paths fragment rankings across duplicate URLs; a lowercasing 301 rule consolidates them, and this models the transform before it ships in server config.

⚡ Mental Model

Like a filing clerk who relabels every folder in lowercase so two spellings never end up in separate drawers.

Project 128 Apply a Redirect Map to a URL List

💡 The Idea Behind It

After a migration you hold a map of old URLs to new ones, and applying it rewrites every known old URL while leaving the rest untouched. This code stores the redirect_map as a dictionary and walks a list of URLs, using get to look up each one. A hit prints an arrow to the new target; a miss prints the URL with a no redirect note. It is the lookup at the heart of a migration rollout.

What You'll Build & Learn

Apply a redirect map to a list of URLs, rewriting mapped ones and passing untracked URLs through unchanged.

- A redirect map drives migrations
- dict.get returns None on a miss
- Untracked URLs pass through unchanged

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 127: redirect mixed-case urls to lowercase. Project 129 changes the surface problem while preserving the underlying method.

progress check

Reading target: apply a redirect map to a url list is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

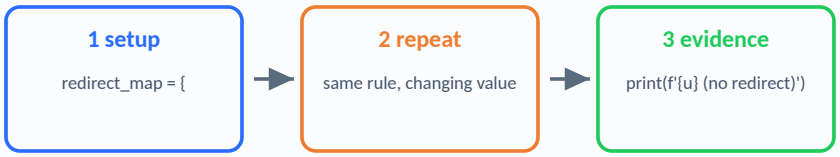
🔎 Logic Lens

- Setup: `redirect_map = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from

302 from canonical.

- Evidence: `print(f'{u} (no redirect)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`redirect_map = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if target:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{u} (no redirect)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if target:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Apply a Redirect Map to a URL List: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: redirect_map, https, old, urls, keep. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: redirect_map = { | Core: if target: | Evidence: print(f'{u} (no redirect)')

🛠️ Reader to Writer Bridge

1. Read: explain apply a redirect map to a url list in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 127.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 129 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{u} (no redirect)')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `redirect_map = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of apply a redirect map to a url list and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A dict lookup with `.get` returns the mapped target when one exists and None otherwise, so unmapped URLs pass through unchanged.

```
redirect_map = {
    'https://old/a': 'https://new/a',
    'https://old/b': 'https://new/b',
}
urls = ['https://old/a', 'https://keep/c', 'https://old/b']
for u in urls:
    target = redirect_map.get(u)
    if target:
        print(f'{u} -> {target}')
    else:
        print(f'{u} (no redirect)')
```

```
output
https://old/a -> https://new/a
https://keep/c (no redirect)
https://old/b -> https://new/b
```

How it works:

1. redirect_map maps two old URLs to their new destinations.
2. urls includes both mapped old URLs plus one that is absent from the map.
3. The loop calls redirect_map.get(u), which returns the target or None.
4. A truthy target prints an arrow to the new URL, otherwise it prints no redirect.

Try it:

1. Report which mapped URLs in the map were never seen in the list.
2. Chase chained map entries so an old target that is itself a key resolves fully.
3. Count how many URLs in the list had no redirect entry.
4. Add a chained entry where a target is also a key and resolve it fully.

⚠ Common Mistakes

- Using `redirect_map[u]` raises KeyError for any URL not in the map.
- Mapping a URL to itself creates a pointless self-redirect.
- Forgetting that a target could itself be a key (a chain) leaves a multi-hop redirect unresolved.

Skills gained redirect map | dict.get lookup | migration rollout

→ Where You'll Use This: Site migrations ship a redirect map; applying it across a URL inventory verifies every old path is covered before launch and shows which URLs still lack a target.

Mental Model

Like a change-of-address book the post office consults for each letter, forwarding the listed ones and delivering the rest as addressed.

Project 129 Find Broken Internal Links

The Idea Behind It

Internal links should point at live pages, and this code cross-checks each link target against the crawl's status map. status holds the code for every crawled URL, links lists source-to-target pairs, and a comprehension keeps any pair whose target status is 404. The output prints each broken link and the total count, surfacing internal links that lead users into a missing page.

What You'll Build & Learn

Find internal links whose target URL returns 404 by cross-referencing each link against the crawl's status map.

- Internal links should reach live pages
- Joining link pairs to a status map
- Comprehension filtering broken targets

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 128: apply a redirect map to a url list. Project 130 changes the surface problem while preserving the underlying method.

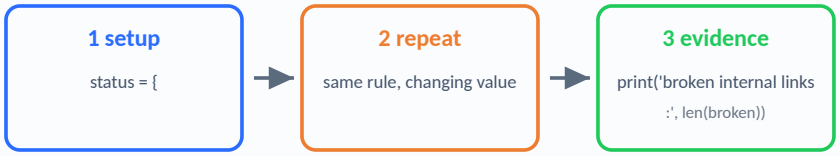
progress check

Reading target: find broken internal links is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `status = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('broken internal links:', len(broken))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`status = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`broken = [(s, t) for s, t in links if status.get(t) == 404]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('broken internal links:', len(broken))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `broken = [(s, t) for s, t in links if status.get(t) == 404]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Broken Internal Links: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: status, https, links, broken, get. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: status = { | Core: broken = [(s, t) for s, t in links if status.get(t) == 404] | Evidence: print('broken internal links:', len(broken))

🛠️ Reader to Writer Bridge

1. Read: explain find broken internal links in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 128.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 130 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('broken internal links:', len(broken))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `status = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find broken internal links and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Looking up each link's target in the status map and keeping those that return 404 isolates internal links pointing at missing pages.

```
status = {
    'https://a/': 200, 'https://b/': 200,
    'https://c/': 404, 'https://d/': 200,
}
links = [
    ('https://a/', 'https://b/'),
    ('https://a/', 'https://c/'),
```

```
    ('https://b/', 'https://d/'),
]
broken = [(s, t) for s, t in links if status.get(t) == 404]
for s, t in broken:
    print(f'{s} -> {t} (404)')
print('broken internal links:', len(broken))
```

```
output
https://a/ -> https://c/ (404)
broken internal links: 1
```

How it works:

1. status maps four crawled URLs to their codes, one of which is a 404.
2. links is a list of source-to-target tuples representing internal links.
3. A comprehension keeps the pairs whose target status, looked up with get, equals 404.
4. The loop prints each broken link, then prints the total broken count.

Try it:

1. Also flag links whose target returns any 4xx or 5xx status.
2. Group the broken links by their source page in the output.
3. Skip targets missing from the status map rather than treating them as live.
4. Also flag links whose target is a 5xx.

⚠ Common Mistakes

- Checking the source page's status instead of the target's flags the wrong end of the link.
- Using `status[t]` raises KeyError for a target not in the crawl map.
- Treating a 410 target as fine ignores another flavour of dead link.

Skills gained broken link detection | status map join | comprehension filter

→ Where You'll Use This: Internal-link audits run exactly this join, flagging links that point at 404s so editors can repoint or remove them before crawlers waste budget following them.

Mental Model

Like checking every signpost in a building against the room directory and noting which ones point to a room that no longer exists.

Project 130 Build a Status-Code Histogram

The Idea Behind It

A crawl returns thousands of statuses, and a histogram of code-to-count gives the shape of a site's health at a glance. This code feeds a list of statuses to Counter, then iterates the sorted unique codes and prints each beside its frequency. The output reads like the summary line every crawl report opens with, condensing raw responses into a tally per distinct code.

What You'll Build & Learn

Build a per-code frequency histogram of crawl statuses using Counter and print it sorted.

- Counter tallies occurrences automatically
- Histogram shows site health shape
- Sorting keys for ordered output

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 129: find broken internal links. Project 131 changes the surface problem while preserving the underlying method.

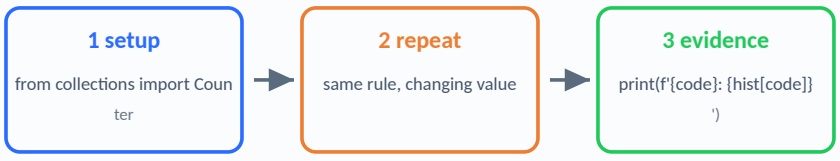
progress check

Reading target: build a status-code histogram is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{code}: {hist[code]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`for code in sorted(hist):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{code}: {hist[code]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for code in sorted(hist):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a Status-Code Histogram: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: collections, Counter, statuses, hist, code. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: for code in sorted(hist): | Evidence: print(f'{code}: {hist[code]}')

🛠️ Reader to Writer Bridge

1. Read: explain build a status-code histogram in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 129.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 131 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code}: {hist[code]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a status-code histogram and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `Counter` tallies each status code in one pass; sorting the keys before printing makes the histogram deterministic and easy to scan.

```
from collections import Counter
statuses = [200, 200, 301, 200, 404, 301, 200, 500, 404, 200]
hist = Counter(statuses)
for code in sorted(hist):
    print(f'{code}: {hist[code]}')
```

```
output
200: 5
301: 2
404: 2
500: 1
```

How it works:

1. A list of ten statuses with repeats is imported into a Counter as hist.
2. Counter builds a mapping from each distinct code to how many times it appeared.
3. The loop iterates the sorted distinct codes.
4. It prints each code beside its count, yielding an ordered histogram.

Try it:

1. Print a simple bar of hash characters proportional to each count.
2. Sort by frequency descending rather than by code.
3. Show each code's percentage share of the total alongside its count.
4. Print the most common code with `hist.most_common(1)`.

⚠ Common Mistakes

- Iterating an unsorted Counter prints the codes in insertion order, which varies with the data.
- Counting class buckets here would lose the per-code detail a histogram is meant to show.
- Summing the counts and forgetting they should equal the crawl size hides dropped responses.

Skills gained status histogram | collections.Counter | sorted keys

→ Where You'll Use This: Crawl dashboards lead with a per-code histogram so an SEO sees at a glance how many 200s, 301s, and 404s a site returns before drilling into specifics.

Mental Model

Like a tally chart that turns a stack of survey forms into a count beside each possible answer.

Project 131 Tally Permanent Versus Temporary Redirects

The Idea Behind It

Permanent and temporary redirects carry different SEO meaning, and the mix tells you whether a site moves URLs decisively. This code defines permanent as the set 301 and 308 and temporary as 302 and 307, then counts how many codes fall in each using generators inside sum. The output reports four permanent against three temporary, splitting the redirects by their intent.

What You'll Build & Learn

Tally permanent versus temporary redirects to gauge how decisively a site moves its URLs.

- 301/308 permanent, 302/307 temporary
- Set membership for classification
- Counting each group with sum

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 130: build a status-code histogram. Project 132 changes the surface problem while preserving the underlying method.

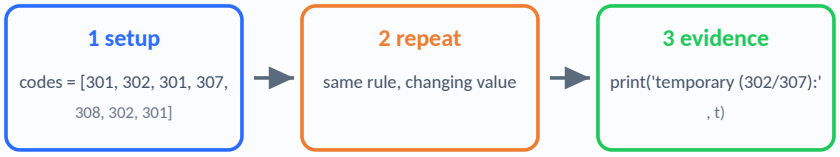
progress check

Reading target: tally permanent versus temporary redirects is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `codes = [301, 302, 301, 307, 308, 302, 301]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('temporary (302/307):', t)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`codes = [301, 302, 301, 307, 308, 302, 301]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`p = sum(1 for c in codes if c in permanent)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('temporary (302/307):', t)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `p = sum(1 for c in codes if c in permanent)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Tally Permanent Versus Temporary Redirects: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: codes, permanent, temporary, sum. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: codes = [301, 302, 301, 307, 308, 302, 301] | Core: p = sum(1 for c in codes if c in permanent) | Evidence: print('temporary (302/307):', t)

🛠️ Reader to Writer Bridge

1. Read: explain tally permanent versus temporary redirects in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 130.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 132 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('temporary (302/307):', t)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `codes = [301, 302, 301, 307, 308, 302, 301]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of tally permanent versus temporary redirects and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Two sets define which codes count as permanent versus temporary, and a pair of generator-sum tallies splits the redirect list by intent.

```
codes = [301, 302, 301, 307, 308, 302, 301]
permanent = {301, 308}
temporary = {302, 307}
p = sum(1 for c in codes if c in permanent)
t = sum(1 for c in codes if c in temporary)
print('permanent (301/308):', p)
print('temporary (302/307):', t)
```

```
output
permanent (301/308): 4
temporary (302/307): 3
```

How it works:

1. codes is a list of seven redirect statuses mixing all four redirect types.
2. permanent is the set {301, 308} and temporary is {302, 307}.
3. p sums one for each code in permanent and t sums one for each in temporary.
4. The code prints the permanent count and the temporary count on separate lines.

Try it:

1. Flag any code that is neither permanent nor temporary as unexpected.
2. Report the ratio of permanent to temporary as a single number.
3. Add 303 and 300 and decide how to classify them.
4. Add a 304 and confirm it counts as neither.

⚠ Common Mistakes

- Forgetting 308 leaves a permanent redirect counted as neither.
- Putting 307 in the permanent set inverts its temporary meaning.
- Counting any 3xx (like 304) as a redirect type pollutes the tally.

Skills gained redirect tally | set membership | sum counting

→ Where You'll Use This: Audits report the permanent-to-temporary split because a site heavy with 302s may be leaking equity on moves that should be permanent 301s.

Mental Model

Like sorting forwarding orders into permanent moves and short holiday holds to see how settled an address book is.

Project 132 Print a Redirect Chain Hop by Hop

The Idea Behind It

Debugging a redirect is easier when you see each hop's status and target on its own line. This printer uses a while True loop: when the current response is a 301 or 302 it prints the URL, its status, and the location it points to, then advances; otherwise it prints the final URL and status and breaks. The result is a readable hop-by-hop trace of the whole chain.

What You'll Build & Learn

Print a redirect chain hop by hop, showing each URL's status and target on its own line.

- Hop-by-hop tracing aids debugging
- while True with an internal break
- Distinguishing a hop from the final stop

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 131: tally permanent versus temporary redirects. Project 133 changes the surface problem while preserving the underlying method.

progress check

Reading target: print a redirect chain hop by hop is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

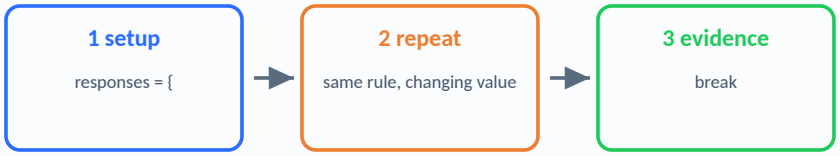
- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `break` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if r['status'] in (301, 302):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`break`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if r['status'] in (301, 302):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Print a Redirect Chain Hop by Hop: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, cur. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: if r['status'] in (301, 302): | Evidence: break

🛠️ Reader to Writer Bridge

1. Read: explain print a redirect chain hop by hop in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 131.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 133 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `break`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of print a redirect chain hop by hop and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The loop prints the current URL with its status, and either follows the `location` of a redirect or marks the terminal page and stops.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://b/'},
    'https://b/': {'status': 302, 'location': 'https://c/'},
    'https://c/': {'status': 200, 'location': None},
}
cur = 'https://a/'
while True:
    r = responses[cur]
```

```
    if r['status'] in (301, 302):
        print(f"{cur} [{r['status']}] -> {r['location']}")
        cur = r['location']
    else:
        print(f"{cur} [{r['status']}] (final)")
        break
```

```
output
https://a/ [301] -> https://b/
https://b/ [302] -> https://c/
https://c/ [200] (final)
```

How it works:

1. The responses dictionary chains a 301 and a 302 into a final 200.
2. cur starts at the first URL inside an unconditional while True loop.
3. When the status is 301 or 302 it prints the URL, status, and location, then advances cur.
4. Otherwise it prints the final URL and status with a final marker and breaks.

Try it:

1. Number each hop with an index as it prints.
2. Add a hop limit so a loop cannot run the trace forever.
3. Include the reason phrase from project 112 beside each status.
4. Add a fourth hop and watch one more line print.

⚠ Common Mistakes

- Printing the target before reading its own status mixes up which hop a code belongs to.
- An infinite `while True` with no terminal branch loops forever on a cyclic chain.
- Using `cur = r['location']` when location is None for a non-redirect would crash the next read.

Skills gained redirect tracing | while True loop | break statement

→ Where You'll Use This: Engineers paste a hop-by-hop trace into bug reports to show exactly where a redirect chain misbehaves, mirroring what curl's verbose follow output reveals.

⚡ Mental Model

Like reading a parcel's tracking history one scan at a time instead of just seeing the final delivery.

Project 133 Check for an Insecure-Protocol Redirect Hop

💡 The Idea Behind It

A redirect chain should not bounce through an insecure step, so this code checks every hop's protocol. is_secure returns whether a URL starts with https://, and a comprehension collects any hop that fails the test. The sample chain hides an insecure:// hop between two https URLs, so the code prints the insecure hops and declares MIXED PROTOCOL, flagging a security and ranking risk.

What You'll Build & Learn

Detect an insecure, non-HTTPS hop inside a redirect chain and flag the mixed-protocol risk.

- Chains must stay on HTTPS throughout
- startswith tests the protocol
- Comprehension collects insecure hops

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 132: print a redirect chain hop by hop. Project 134 changes the surface problem while preserving the underlying method.

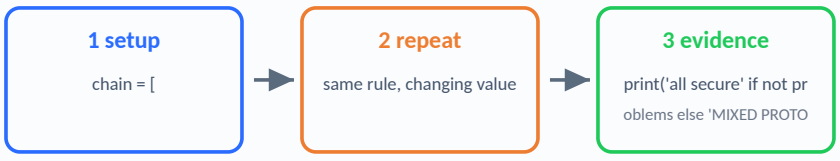
progress check

Reading target: check for an insecure-protocol redirect hop is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `chain = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('all secure' if not problems else 'MIXED PROTOCOL')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`chain = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`problems = [u for u in chain if not is_secure(u)]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('all secure' if not problems else 'MIXED PROTOCOL')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `problems = [u for u in chain if not is_secure(u)]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Check for an Insecure-Protocol Redirect Hop: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: chain, https, insecure, is_secure, startswith. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: chain = [| Core: problems = [u for u in chain if not is_secure(u)] | Evidence: print('all secure' if not problems else 'MIXED PROTOCOL')

🛠️ Reader to Writer Bridge

1. Read: explain check for an insecure-protocol redirect hop in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 132.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 134 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('all secure' if not problems else 'MIXED PROTOCOL')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `chain = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check for an insecure-protocol redirect hop and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Testing each hop's prefix against the secure scheme isolates any step that falls back to an insecure protocol, which weakens the whole chain.

```
chain = [
    'https://a/',
    'insecure://a/',
    'https://b/',
]
def is_secure(u):
    return u.startswith('https://')
problems = [u for u in chain if not is_secure(u)]
print('insecure hops:', problems if problems else 'none')
print('all secure' if not problems else 'MIXED PROTOCOL')
```

```
output
insecure hops: ['insecure://a/']
MIXED PROTOCOL
```

How it works:

1. chain is a list of three URLs where the middle one uses an insecure protocol.

2. is_secure returns True only when a URL starts with https://.
3. A comprehension collects every URL that fails the secure test into problems.
4. The code prints the insecure hops, then all secure or MIXED PROTOCOL depending on whether problems is empty.

Try it:

1. Report the position index of each insecure hop in the chain.
2. Treat http:// specifically and suggest the https:// upgrade target.
3. Fail only when the insecure hop is not the very first entry.
4. Make every hop secure and confirm the 'all secure' verdict.

⚠ Common Mistakes

- Checking only the first and last URL misses an insecure hop in the middle.
- A prefix test that accepts any scheme containing 'https' could be fooled by a lookalike host.
- Treating a protocol-relative URL (`//host`) as secure assumes the page context, which may be wrong.

Skills gained protocol check | startswith test | comprehension filter

→ Where You'll Use This: Audits verify redirect chains never drop to HTTP mid-flight; an insecure hop triggers browser warnings and weakens trust signals, so flagging it is a security and SEO requirement.

Mental Model

Like noticing a secure courier route briefly hands the package to an unmarked van before resuming.

Project 134 Handle 410 Gone Versus 404 Not Found

The Idea Behind It

404 means missing, maybe temporarily, while 410 means gone for good, and treating them differently tells a crawler whether to keep retrying or retire a URL. This code walks a list of URL and code pairs, mapping 410 to retire permanently, 404 to recrawl later, and anything else to keep. The output assigns each page its action, encoding the gone-versus-not-found distinction.

What You'll Build & Learn

Assign a crawler action per page by distinguishing a permanent 410 Gone from a recheckable 404 Not Found.

- 410 is permanent, 404 may recover
- if-elif-else action selection
- Status drives recrawl behaviour

Skill Roadmap

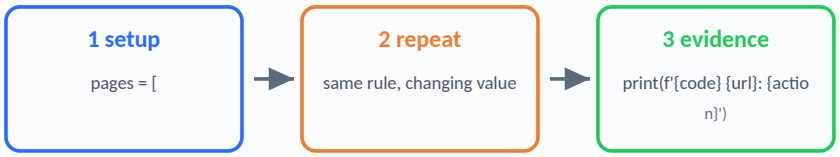
Skill step: moves you from recognition toward independent tracing. It builds on Project 133: check for an insecure-protocol redirect hop. Project 135 changes the surface problem while preserving the underlying method. *progress check*

Reading target: handle 410 gone versus 404 not found is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{code} {url}: {action}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if code == 410:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{code} {url}: {action}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if code == 410:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Handle 410 Gone Versus 404 Not Found: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: pages, https, url, code, action. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = [| Core: if code == 410: | Evidence: print(f'{code} {url}: {action}')

Reader to Writer Bridge

1. Read: explain handle 410 gone versus 404 not found in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 133.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 135 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code} {url}: {action}')`. Then compare punctuation, spacing, and order.
2. Explain: point to `pages = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of handle 410 gone versus 404 not found and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Branching on 410 versus 404 encodes the semantic difference: a 410 lets the crawler drop the URL for good, while a 404 leaves room for it to return.

```
pages = [
    ('https://a/', 404),
    ('https://b/', 410),
    ('https://c/', 404),
    ('https://d/', 200),
]
for url, code in pages:
    if code == 410:
        action = 'retire permanently'
    elif code == 404:
        action = 'recrawl later'
    else:
        action = 'keep'
    print(f'{code} {url}: {action}')
```

```
output
404 https://a/: recrawl later
410 https://b/: retire permanently
404 https://c/: recrawl later
200 https://d/: keep
```

How it works:

1. pages is a list of URL and code tuples covering 404, 410, and 200.
2. The loop assigns retire permanently when the code is 410.
3. It assigns recrawl later for 404 and keep for everything else.
4. An f-string prints the code, URL, and chosen action for each page.

Try it:

1. Add a 451 unavailable-for-legal-reasons branch with its own action.
2. Track how many recrawl-later URLs accumulate across the list.
3. Treat repeated 404s over several crawls as an eventual retire.

4. Add a 403 and decide its action.

⚠ Common Mistakes

- Collapsing 410 into the 404 branch loses the 'permanently gone' signal.
- Treating both as a generic 4xx error skips the chance to stop recrawling 410s.
- Assuming a 200 page needs no action when it might still carry a noindex tag.

Skills gained 410 vs 404 | if-elif-else | recrawl logic

→ Where You'll Use This: Crawl schedulers honour 410 by dropping URLs immediately while retrying 404s; modelling this difference keeps a crawler from wasting requests on pages a site has deliberately removed.

⚡ Mental Model

Like a postal service that re-attempts an unanswered house but permanently closes the file on a demolished one.

Project 135 Interpret a 304 Not Modified Response

💡 The Idea Behind It

A 304 says the cached copy is still fresh, so the server sends no body and the client reuses what it has. This code walks a list of responses carrying a status and a body length; for a 304 it notes use cached copy with no body sent, and for a 200 it reports the fresh byte count. Recognising 304 avoids mistaking an empty response for an error.

What You'll Build & Learn

Interpret a 304 Not Modified as a cache reuse signal rather than treating its empty body as an error.

- 304 means reuse the cached copy
- An empty body on 304 is normal
- Branching on status for the note

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 134: handle 410 gone versus 404 not found. Project 136 changes the surface problem while preserving the underlying method.

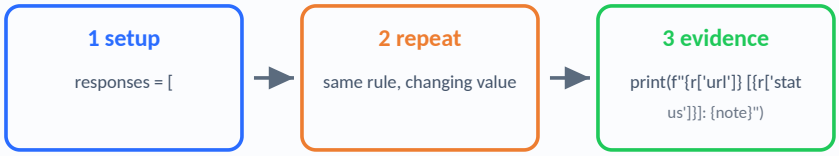
progress check

Reading target: interpret a 304 not modified response is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `responses = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f"{r['url']} [{r['status']}]: {note}")` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if r['status'] == 304:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{r['url']} [{r['status']}]: {note}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if r['status'] == 304:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Interpret a 304 Not Modified Response: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, url, https, status, body_len. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = [| Core: if r['status'] == 304: | Evidence: print(f"{r['url']} [{r['status']}]: {note}")

🛠️ Reader to Writer Bridge

1. Read: explain interpret a 304 not modified response in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 134.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 136 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{r['url']} [{r['status']}]: {note}")`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `responses = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of interpret a 304 not modified response and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A 304 legitimately carries no body because the client's cached version is still valid, so an empty `body_len` is expected rather than a fault.

```
responses = [
    {'url': 'https://a/', 'status': 200, 'body_len': 1500},
    {'url': 'https://b/', 'status': 304, 'body_len': 0},
    {'url': 'https://c/', 'status': 200, 'body_len': 800},
]
for r in responses:
    if r['status'] == 304:
        note = 'use cached copy (no body sent)'
    else:
        note = f"fresh body, {r['body_len']} bytes"
    print(f"{r['url']} [{r['status']}]: {note}")
```

```
output
https://a/ [200]: fresh body, 1500 bytes
https://b/ [304]: use cached copy (no body sent)
https://c/ [200]: fresh body, 800 bytes
```

How it works:

1. A list of response dictionaries holds a url, status, and body_len.
2. The middle entry is a 304 with a body_len of zero.
3. The loop prints use cached copy for a 304 and a fresh-body byte count otherwise.
4. An f-string shows each URL, status, and its note.

Try it:

1. Add the request headers, such as If-None-Match, that elicit a 304.
2. Count how many responses were served from cache versus fresh.
3. Treat a 304 with a non-zero body length as suspicious.
4. Add another 304 and confirm both reuse the cache.

⚠ Common Mistakes

- Flagging the empty body of a 304 as a broken page misreads a normal cache hit.
- Expecting a `body_len` greater than zero on a 304 contradicts the spec.

- Treating 304 as a redirect because it is a 3xx ignores that it carries no `location`.

Skills gained 304 interpretation | conditional notes | cache awareness

→ Where You'll Use This: Crawlers send conditional requests and honour 304 to save bandwidth; understanding that an empty 304 body is expected prevents tools from flagging healthy cached pages as broken.

Mental Model

Like a librarian replying nothing's changed, keep the copy you have rather than handing over the whole book again.

Project 136 Detect a Redirect Pointing to a Missing Entry

The Idea Behind It

A redirect map can point at a URL that was never crawled, leaving the chain unresolved. This walker guards the loop with a membership check: it advances only while the current URL exists in responses and is a 301 or 302. After the loop it tests whether cur is still present; if not, it reports the dangling target. The guard prevents a KeyError mid-walk when a hop leads off the known set.

What You'll Build & Learn

Detect a redirect whose target was never crawled and report it as dangling without raising a KeyError.

- A hop can target an uncrawled URL
- Membership guard prevents KeyError
- Dangling targets stay unresolved

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 135: interpret a 304 not modified response. Project 137 changes the surface problem while preserving the underlying method.

progress check

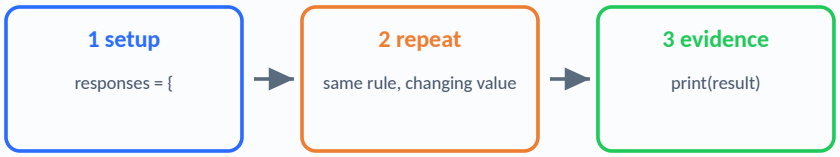
Reading target: detect a redirect pointing to a missing entry is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(result)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if cur in responses:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(result)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if cur in responses:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Redirect Pointing to a Missing Entry: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, ghost. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: if cur in responses: | Evidence: print(result)

🛠️ Reader to Writer Bridge

1. Read: explain detect a redirect pointing to a missing entry in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 135.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 137 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(result)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a redirect pointing to a missing entry and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Guarding the loop with `cur in responses` stops the walk safely when a hop points at a URL absent from the map, which the code then reports as dangling.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://b/'},
    'https://b/': {'status': 301, 'location': 'https://ghost/'},
}
cur = 'https://a/'
result = 'unresolved'
while cur in responses and responses[cur]['status'] in (301, 302):
    cur = responses[cur]['location']
if cur in responses:
    result = f"resolved at {cur} [{responses[cur]['status']}]"
else:
    result = f'DANGLING: {cur} not crawled'
print(result)
```

```
output
DANGLING: https://ghost/ not crawled
```

How it works:

1. The responses dictionary chains two redirects whose final location, a ghost URL, is absent from the map.
2. The while loop advances only while cur is in responses and its status is 301 or 302.
3. After the loop, if cur is still in responses it reports the resolved status.
4. Otherwise it prints DANGLING with the uncrawled target.

Try it:

1. Queue the dangling target for a follow-up crawl pass.
2. Distinguish a dangling external host from an uncrawled internal one.
3. Count how many chains end dangling across a larger map.
4. Add the missing `https://ghost/` entry with status 200 and watch it resolve.

⚠ Common Mistakes

- Dropping the `cur in responses` guard raises KeyError the moment a target is missing.
- Assuming every `location` exists in the map is exactly the bug this guards against.
- Reporting success when the loop exits without confirming the final URL was actually crawled.

Skills gained dangling detection | membership guard | KeyError safety

→ Where You'll Use This: Incomplete crawls leave redirect targets unfetched; guarding lookups lets an audit report unresolved chains gracefully instead of crashing partway through a large URL set.

Mental Model

Like a forwarding chain that points to a street the census never surveyed, so you note the gap rather than insist it exists.

Project 137 Detect a URL That Redirects to Itself

The Idea Behind It

A URL redirecting to itself is the tightest possible loop and an immediate crawl trap. This code scans the sorted response map and, for each URL that is a 301 or 302, checks whether its location equals the URL itself. The first entry points at its own address, so it prints SELF-REDIRECT, while the others read ok. Catching this before any walk begins prevents an instant infinite loop.

What You'll Build & Learn

Flag any URL whose redirect location equals itself, catching the tightest possible loop before a walk.

- A self-redirect is an instant loop
- Comparing location to the URL
- Catching the trap before walking

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 136: detect a redirect pointing to a missing entry. Project 138 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a url that redirects to itself is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

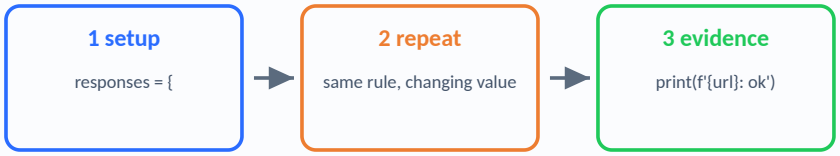
- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{url}: ok')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if r['status'] in (301, 302) and r['location'] == url:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{url}: ok')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if r['status'] in (301, 302) and r['location'] == url:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a URL That Redirects to Itself: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, url. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: if r['status'] in (301, 302) and r['location'] == url: | Evidence: print(f'{url}: ok')

🛠️ Reader to Writer Bridge

1. Read: explain detect a url that redirects to itself in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 136.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 138 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{url}: ok')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a url that redirects to itself and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A self-redirect is the simplest cycle to spot: a redirect status whose `location` is identical to its own URL, caught without following anything.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://a/'},
    'https://b/': {'status': 301, 'location': 'https://c/'},
    'https://c/': {'status': 200, 'location': None},
}
for url in sorted(responses):
    r = responses[url]
    if r['status'] in (301, 302) and r['location'] == url:
```

```
        print(f'{url}: SELF-REDIRECT')
    else:
        print(f'{url}: ok')
```

```
output
https://a/: SELF-REDIRECT
https://b/: ok
https://c/: ok
```

How it works:

1. The responses dictionary includes one URL whose location points back at itself.
2. The loop iterates the sorted URLs and reads each response.
3. It flags SELF-REDIRECT when the status is 301 or 302 and the location equals the URL.
4. Every other URL prints ok.

Try it:

1. Also catch a URL whose location differs only by a trailing slash.
2. Count how many self-redirects appear across the whole map.
3. Report the status code alongside each self-redirect flag.
4. Add a 302 self-redirect and confirm it is also caught.

⚠ Common Mistakes

- Comparing the URLs after normalising one but not the other can miss a slash-only self-loop.
- Only checking 301 misses a 302 self-redirect.
- Iterating an unsorted dict makes the per-URL output order vary.

Skills gained self-redirect | self-comparison | loop prevention

→ Where You'll Use This: A self-redirect hangs browsers and crawlers immediately; scanning the response map for one is a cheap pre-check that catches the worst misconfiguration before a chain walk even starts.

Mental Model

Like a forwarding note on a door that simply reads see this same door, sending you in circles on the spot.

Project 138 Average the Hop Count Across Chains

The Idea Behind It

One long chain is bad, but the site-wide average tells you whether redirects are a systemic problem. This code stores each chain's pre-counted hops in a dictionary, pulls the values into a list, sums them, and divides by the

count to get a rounded average. The output reports the number of chains, total hops, and the average, summarising the whole redirect map in a single figure.

What You'll Build & Learn

Average the hop counts across all redirect chains to judge whether redirects are a systemic issue.

- Average reveals systemic redirect debt
- dict.values feeds a sum
- Rounding the mean to two decimals

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 137: detect a url that redirects to itself. Project 139 changes the surface problem while preserving the underlying method.

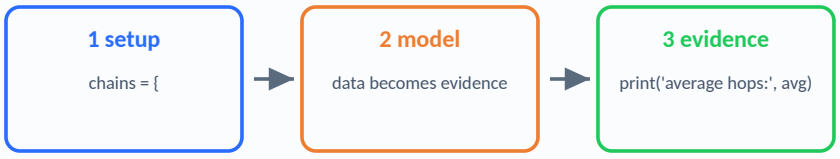
progress check

Reading target: average the hop count across chains is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `chains = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as data -> transformation -> evidence, not as isolated library calls.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('average hops:', avg)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, model applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`chains = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`print('average hops:', avg)`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print('average hops:', avg)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('average hops:', avg)`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Average the Hop Count Across Chains: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: chains, https, hops, values, avg. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: chains = { | Core: print('average hops:', avg) | Evidence: print('average hops:', avg)

🛠️ Reader to Writer Bridge

1. Read: explain average the hop count across chains in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 137.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 139 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('average hops:', avg)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `chains = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of average the hop count across chains and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Summing the per-chain hop counts and dividing by the number of chains, then rounding, gives a single comparable redirect-depth metric.

```
chains = {
    'https://a/': 1,
    'https://b/': 3,
    'https://c/': 2,
    'https://d/': 0,
}
hops = list(chains.values())
avg = round(sum(hops) / len(hops), 2)
print('chains:', len(hops))
print('total hops:', sum(hops))
print('average hops:', avg)
```

```
output
chains: 4
total hops: 6
average hops: 1.5
```

How it works:

1. chains maps four starting URLs to their pre-counted hop totals.

2. hops collects the dictionary's values into a list.
3. avg divides the sum of hops by their count and rounds to two decimals.
4. The code prints the chain count, total hops, and the average.

Try it:

1. Also report the maximum and minimum hop count across the chains.
2. Exclude zero-hop chains from the average and compare the result.
3. Compute the median hop count alongside the mean.
4. Drop the zero-hop chain and recompute the average.

⚠ Common Mistakes

- Including chains with zero hops or excluding them changes the average; be explicit about which you mean.
- Integer division would truncate the average to a whole number.
- Forgetting `round` prints a long repeating float.

Skills gained average hops | dict.values | mean rounding

→ Where You'll Use This: Tracking the average hop count across crawls shows whether a redirect cleanup is working; a falling mean signals the site is consolidating chains rather than accumulating them.

Mental Model

Like reporting a fleet's average detour length rather than fixating on the one truck that took the longest way round.

Project 139 Resolve a Redirect Map to Final Targets

The Idea Behind It

A migration map often has chained entries where one target is itself a key, so following it once is not enough. The final function walks the map until the URL is no longer a key, using a seen set to stop if it revisits a URL. The result gives every old URL its true final destination in one pass, even when the map links a to b to c.

What You'll Build & Learn

Resolve a redirect map with chained entries to each URL's true final target, guarding against cycles.

- Map entries can chain together
- Following until the key stops
- A seen set prevents infinite loops

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 138: average the hop count across chains. Project 140 changes the surface problem while preserving the underlying method.

progress check

Reading target: resolve a redirect map to final targets is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `redirect_map = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{start} => {final(start)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`redirect_map = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`while url in redirect_map and url not in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{start} => {final(start)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `while url in redirect_map and url not in seen:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Resolve a Redirect Map to Final Targets: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: redirect_map, https, url, seen, not. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: redirect_map = { | Core: while url in redirect_map and url not in seen: | Evidence: print(f'{start} => {final(start)}')

Reader to Writer Bridge

1. Read: explain resolve a redirect map to final targets in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 138.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 140 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{start} => {final(start)}')`. Then compare punctuation, spacing, and order.
2. Explain: point to `redirect_map = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of resolve a redirect map to final targets and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Repeatedly replacing a URL with its mapped target until the map runs out (with a seen-set loop guard) flattens any chain into a single final hop.

```
redirect_map = {
    'https://a/': 'https://b/',
    'https://b/': 'https://c/',
    'https://d/': 'https://e/',
}
def final(url, seen=None):
    seen = seen or set()
    while url in redirect_map and url not in seen:
        seen.add(url)
        url = redirect_map[url]
    return url
for start in sorted(redirect_map):
    print(f'{start} => {final(start)}')
```

```
output
https://a/ => https://c/
https://b/ => https://c/
https://d/ => https://e/
```

How it works:

1. redirect_map links a to b to c, with d pointing separately to e.
2. final loops while the URL is still a key and not already in seen.
3. Each step adds the URL to seen and follows the map to the next.
4. The loop prints each starting URL alongside its fully resolved final target.

Try it:

1. Return the full path taken, not just the final target.
2. Report any starting URL that loops back into a cycle.
3. Use the resolved targets to rewrite the map into single hops.

4. Add a cyclic entry and confirm the guard stops it.

⚠ Common Mistakes

- Returning the first target instead of following the chain leaves a multi-hop redirect.
- A default mutable argument shared across calls would accumulate seen URLs; building it per call avoids that.
- Without the seen-set guard a cyclic map would loop forever.

Skills gained map resolution | chained lookup | cycle guard

→ Where You'll Use This: Flattening a chained migration map gives clean one-hop redirects; resolving each old URL to its final target lets an SEO replace multi-step chains with direct 301s that preserve more equity.

⚡ Mental Model

Like following a chain of forwarding addresses until you reach the one nobody has moved on from.

Project 140 Summarise a Crawl by Status Class

💡 The Idea Behind It

Rolling thousands of codes up into 2xx, 3xx, 4xx, and 5xx buckets is the headline a crawl report opens with. This code feeds a generator that formats each status as a family label into Counter, producing a count per family. Iterating the sorted labels prints the four numbers, compressing a full per-code histogram into a single line of site health.

What You'll Build & Learn

Summarise a crawl by counting responses into 2xx, 3xx, 4xx, and 5xx family buckets.

- Family buckets compress the histogram
- Generator builds the class label
- Counter tallies the four families

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 139: resolve a redirect map to final targets. Project 141 changes the surface problem while preserving the underlying method.

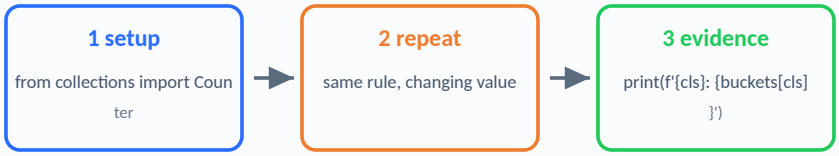
progress check

Reading target: summarise a crawl by status class is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{cls}: {buckets[cls]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`buckets = Counter(f'{s // 100}xx' for s in statuses)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{cls}: {buckets[cls]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `buckets = Counter(f'{s // 100}xx' for s in statuses)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Summarise a Crawl by Status Class: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: collections, Counter, statuses, buckets, xx. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: buckets = Counter(f'{s // 100}xx' for s in statuses) | Evidence: print(f'{cls}: {buckets[cls]}')

🛠️ Reader to Writer Bridge

1. Read: explain summarise a crawl by status class in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 139.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 141 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{cls}: {buckets[cls]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of summarise a crawl by status class and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Mapping each code to its `Nxx` class string with integer division and counting those strings produces the four-bucket summary directly.

```
from collections import Counter
statuses = [200, 301, 200, 404, 500, 302, 200, 403, 503, 200]
buckets = Counter(f'{s // 100}xx' for s in statuses)
for cls in sorted(buckets):
    print(f'{cls}: {buckets[cls]}')
```

```
output
2xx: 4
3xx: 2
4xx: 2
5xx: 2
```

How it works:

1. statuses is a list of ten codes spanning all four families.
2. A generator formats each status as its family label, like 2xx, via integer division.
3. Counter tallies those labels into buckets.
4. The loop prints each sorted family label beside its count.

Try it:

1. Add a percentage share for each family alongside the count.
2. Include a 1xx bucket and feed it an informational code.
3. List the actual URLs under each family using project 145's approach.
4. Print each bucket as a share of the total.

⚠ Common Mistakes

- Bucketing on `str(s)[0]` works for three-digit codes but is fragile if a code is malformed.
- Forgetting to sort the buckets prints the classes in arbitrary order.
- Treating 304 as an error rather than a 3xx redirect-class code miscounts the buckets.

Skills gained class summary | Counter with generator | family bucketing

→ Where You'll Use This: The four-family rollup is the top-line metric of every crawl report; it lets a stakeholder gauge overall site health in one glance before any URL-level detail.

Mental Model

Like summarising a city's weather report as days of sun, cloud, rain, and storm rather than every hourly reading.

Project 141 Flag a Redirect Chain That Leaves the Domain

The Idea Behind It

A chain that hops to an external domain may signal a hijack or a sloppy migration. The host helper extracts the hostname by splitting off the scheme and then the path. The code takes the first URL's host as the origin and gathers any hop whose host differs. The sample jumps to other.com, so the code prints the origin host and the off-domain hops it found.

What You'll Build & Learn

Flag a redirect chain that leaves its origin domain by comparing each hop's host to the first one.

- Off-domain hops can signal a hijack
- Extracting a host from a URL
- Comparing each hop's host to origin

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 140: summarise a crawl by status class. Project 142 changes the surface problem while preserving the underlying method.

progress check

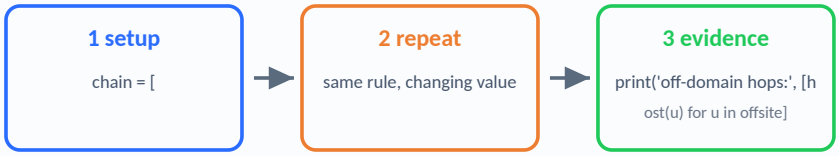
Reading target: flag a redirect chain that leaves the domain is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `chain = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('off-domain hops:', [host(u) for u in offsite] if offsite else 'none')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`chain = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`offsite = [u for u in chain if host(u) != origin]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('off-domain hops:', [host(u) for u in offsite] if offsite else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `offsite = [u for u in chain if host(u) != origin]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Flag a Redirect Chain That Leaves the Domain: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: chain, https, shop, com, other. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: chain = [| Core: offsite = [u for u in chain if host(u) != origin] | Evidence: print('off-domain hops:', [host(u) for u in offsite] if offsite else 'n...

🛠️ Reader to Writer Bridge

1. Read: explain flag a redirect chain that leaves the domain in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 140.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 142 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('off-domain hops:', [host(u) for u in offsite] if offsite else 'none')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `chain = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag a redirect chain that leaves the domain and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Extracting each hop's host and comparing it to the first hop's host isolates any step that leaves the origin domain.

```
chain = [
    'https://shop.com/a',
    'https://shop.com/b',
    'https://other.com/c',
```

```
]
def host(u):
    return u.split('://', 1)[1].split('/', 1)[0]
origin = host(chain[0])
offsite = [u for u in chain if host(u) != origin]
print('origin host:', origin)
print('off-domain hops:', [host(u) for u in offsite] if offsite else 'none')
```

```
output
origin host: shop.com
off-domain hops: ['other.com']
```

How it works:

1. chain lists three URLs where the last belongs to a different domain.
2. host splits a URL on :// then on the first slash to isolate the hostname.
3. origin is the host of the first URL in the chain.
4. A comprehension collects hops whose host differs from origin, then the code prints the origin and any off-domain hosts.

Try it:

1. Allow a list of trusted sibling domains that should not be flagged.
2. Treat a subdomain of the origin as on-domain rather than off.
3. Report the index of the first off-domain hop in the chain.
4. Point the last hop back to `shop.com` and confirm 'none'.

⚠ Common Mistakes

- Splitting only on `://` and forgetting to cut at the first `/` keeps the path in the host.
- Comparing full URLs instead of hosts flags every path change as off-domain.
- Ignoring a port or subdomain difference may over- or under-flag depending on intent.

Skills gained off-domain detection | host extraction | comprehension filter

→ Where You'll Use This: Audits flag chains that drift to foreign domains because they often indicate a compromised redirect or a botched migration; comparing hosts is the check that surfaces the unexpected jump.

Mental Model

Like noticing a forwarded letter suddenly routes through another country before claiming to come back home.

Project 142 Decide Whether a Status Is Retryable

The Idea Behind It

A crawler should retry transient failures but give up on permanent ones. This code defines a retryable set of codes (429 and the common 5xx failures), then classifies each status: a 2xx is done, a member of the retryable set is retry later, and everything else is give up. The output drives a fetch queue's retry logic, deciding which requests deserve another attempt.

What You'll Build & Learn

Classify each status as done, retry later, or give up to drive a fetch queue's retry decisions.

- Transient failures deserve a retry
- A retryable set encodes the policy
- if-elif-else assigns the verdict

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 141: flag a redirect chain that leaves the domain. Project 143 changes the surface problem while preserving the underlying method.

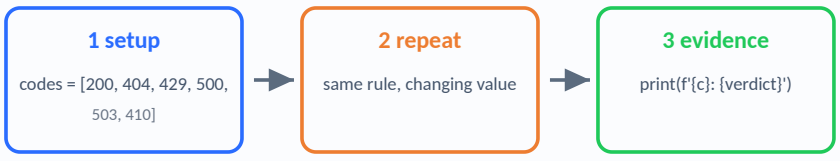
progress check

Reading target: decide whether a status is retryable is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `codes = [200, 404, 429, 500, 503, 410]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{c}: {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`codes = [200, 404, 429, 500, 503, 410]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if 200 <= c < 300:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{c}: {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if 200 <= c < 300:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Decide Whether a Status Is Retryable: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: codes, retryable, verdict, done, retry. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: codes = [200, 404, 429, 500, 503, 410] | Core: if 200 <= c < 300: | Evidence: print(f'{c}: {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain decide whether a status is retryable in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 141.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 143 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c}: {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `codes = [200, 404, 429, 500, 503, 410]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of decide whether a status is retryable and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Success codes finish, a fixed set of transient codes (rate-limit and 5xx) are retried, and everything else is treated as a permanent failure.

```
codes = [200, 404, 429, 500, 503, 410]
retryable = {429, 500, 502, 503, 504}
for c in codes:
    if 200 <= c < 300:
        verdict = 'done'
    elif c in retryable:
        verdict = 'retry later'
    else:
        verdict = 'give up'
    print(f'{c}: {verdict}')
```

```
output
200: done
404: give up
429: retry later
500: retry later
503: retry later
410: give up
```

How it works:

1. codes lists a spread of statuses from 200 through 503 and 410.
2. retryable is a set of transient codes including 429 and several 5xx.
3. The loop assigns done for any 2xx, retry later for a retryable code, and give up otherwise.
4. An f-string prints each code beside its verdict.

Try it:

1. Add a backoff delay that grows with each retry attempt.
2. Cap the number of retries before a retryable code becomes give up.
3. Treat a 403 differently from a 404 in the give-up branch.
4. Add 501 and decide whether it belongs in the retry set.

⚠ Common Mistakes

- Retrying a 404 wastes budget on a page that will not appear by retrying.
- Treating every 5xx as retryable includes 501 (not implemented), which usually will not change.
- Forgetting 429 (too many requests) means ignoring a server explicitly asking you to slow down.

Skills gained retry policy | set membership | if-elif-else

→ Where You'll Use This: Crawl fetchers consult this kind of policy to avoid hammering permanently broken URLs while still retrying rate-limited or temporarily failing ones, keeping crawl budget focused.

⚡ Mental Model

Like a delivery driver who knocks again at a busy house but crosses off an address that no longer exists.

Project 143 Spot a Redirect That Only Strips Query Params

💡 The Idea Behind It

A common redirect drops tracking parameters, sending /page?ref=x to /page. This code compares each redirect's source path, stripped of its query, against the destination. When they match, it is a query-strip that canonicalises rather than moves the page; when they differ, it is a genuine page-move. The output labels each redirect, confirming whether it merely cleans parameters or relocates content.

What You'll Build & Learn

Tell a query-stripping canonicalisation redirect apart from a real page-move by comparing paths.

- Query-strip redirects canonicalise URLs
- Splitting off the query string
- Path equality distinguishes the kinds

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 142: decide whether a status is retryable. Project 144 changes the surface problem while preserving the underlying method.

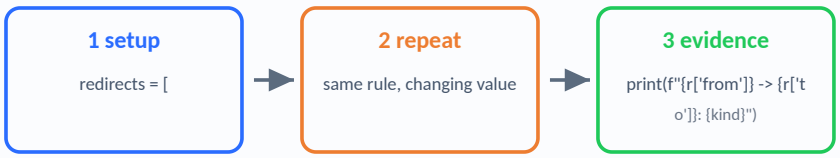
progress check

Reading target: spot a redirect that only strips query params is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `redirects = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f"{r['from']} -> {r['to']}: {kind}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`redirects = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`kind = 'query-strip' if same_path else 'page-move'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{r['from']} -> {r['to']}: {kind}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `kind = 'query-strip' if same_path else 'page-move'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Spot a Redirect That Only Strips Query Params: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: redirects, https, ref, ad, to. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: redirects = [| Core: kind = 'query-strip' if same_path else 'page-move' | Evidence: print(f"{r['from']} -> {r['to']}: {kind}")

Reader to Writer Bridge

1. Read: explain spot a redirect that only strips query params in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 142.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 144 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{r['from']} -> {r['to']}: {kind}")`. Then compare punctuation, spacing, and order.
2. Explain: point to `redirects = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of spot a redirect that only strips query params and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Comparing the source URL with its query removed against the target reveals whether the redirect merely strips parameters or actually moves to a different page.

```
redirects = [
    {'from': 'https://s/p?ref=ad', 'to': 'https://s/p'},
    {'from': 'https://s/a', 'to': 'https://s/b'},
]
def path_only(u):
    return u.split('?', 1)[0]
for r in redirects:
    same_path = path_only(r['from']) == r['to']
    kind = 'query-strip' if same_path else 'page-move'
    print(f"{r['from']} -> {r['to']}: {kind}")
```

```
output
https://s/p?ref=ad -> https://s/p: query-strip
https://s/a -> https://s/b: page-move
```

How it works:

1. redirects lists one query-stripping redirect and one that moves to a different page.
2. path_only splits a URL on the question mark and keeps the part before it.
3. same_path is True when the source's path equals the destination exactly.
4. The loop labels each redirect query-strip or page-move and prints it.

Try it:

1. Print which query parameters were removed by the strip.
2. Treat a redirect that adds parameters as its own category.
3. Ignore harmless parameters like a session id when comparing paths.
4. Add a redirect that strips only one of two params and decide how to classify it.

⚠ Common Mistakes

- Comparing the full source URL to the target never matches because the query differs.
- Splitting on `?` and keeping the second half inspects the params instead of the path.
- A target that itself has a query would break the equality even for a pure strip.

Skills gained query stripping | path comparison | redirect classification

→ Where You'll Use This: Distinguishing canonicalising redirects from content moves helps an audit understand intent; a query-strip is healthy housekeeping, while a page-move needs equity-transfer scrutiny.

Mental Model

Like spotting whether a form simply erased a scribbled note in the margin or rewrote the whole address.

Project 144 Validate That Canonicals Point at Live Pages

The Idea Behind It

A canonical tag is wasted if it points at a URL that redirects or 404s. This code holds a status map and a dictionary of page-to-canonical targets, then checks each target's status with get, defaulting to 0 when absent. Only a 200 target counts as OK; a 301 target and a 404 target both read BAD canonical. Verifying targets confirms the consolidation actually works.

What You'll Build & Learn

Validate that every canonical tag points at a live 200 page rather than a redirect or error.

- Canonicals must target live 200 pages
- A redirecting canonical is wasted
- Default status on a missing target

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 143: spot a redirect that only strips query params. Project 145 changes the surface problem while preserving the underlying method.

progress check

Reading target: validate that canonicals point at live pages is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

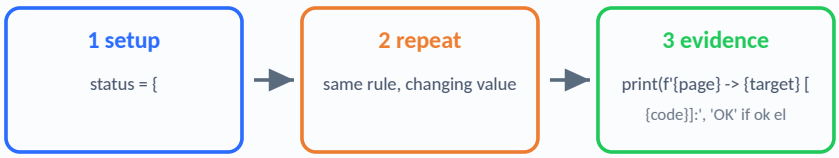
Logic Lens

- Setup: `status = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`status = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate That Canonicals Point at Live Pages: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: status, https, canonicals, page, sorted. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: status = { | Core: print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical') | Evidence: print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical')

🛠️ Reader to Writer Bridge

1. Read: explain validate that canonicals point at live pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 143.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 145 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `status = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate that canonicals point at live pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A canonical is valid only when its target returns 200; pointing at a 301 or 404 means the declared canonical URL is not the live, indexable page.

```
status = {
    'https://a/': 200, 'https://b/': 301,
    'https://c/': 200, 'https://d/': 404,
}
canonicals = {
    'https://a/': 'https://a/',
    'https://c/': 'https://b/',
    'https://x/': 'https://d/',
}
for page in sorted(canonicals):
    target = canonicals[page]
    code = status.get(target, 0)
    ok = code == 200
    print(f'{page} -> {target} [{code}]:', 'OK' if ok else 'BAD canonical')
```

```
output
https://a/ -> https://a/ [200]: OK
https://c/ -> https://b/ [301]: BAD canonical
https://x/ -> https://d/ [404]: BAD canonical
```

How it works:

1. status maps four URLs to codes including a 301 and a 404.
2. canonicals maps each page to the URL its canonical tag declares.
3. The loop looks up each target's status, defaulting to 0 when it is missing.
4. It prints OK only when the target is 200, otherwise BAD canonical.

Try it:

1. Treat a target outside the status map as an explicit unknown.
2. Also flag a canonical that points at another canonical chain.
3. Report how many canonicals across the set are healthy versus broken.
4. Make `https://b/` return 200 and watch that canonical pass.

⚠ Common Mistakes

- Checking the page's own status instead of the canonical target's misses a bad target.
- Treating a canonical that points at a 301 as fine sends engines on an extra hop.
- Using `status[target]` raises KeyError when the target was never crawled.

Skills gained canonical validation | status lookup | default value

→ Where You'll Use This: SEO audits verify canonical targets resolve to 200; a canonical aimed at a redirect or 404 silently fails to consolidate, so this check confirms each tag does its job.

⚡ Mental Model

Like checking that every Master copy label on a file points to a folder that actually exists and is not just another redirect.

Project 145 Group Crawled URLs by Status Class

💡 The Idea Behind It

An audit report lists offending URLs under each status class, not just counts. This code uses a defaultdict of lists, appending each crawled URL to the bucket named by its family label. Iterating the sorted classes prints each family followed by its sorted URLs, producing the body of the report rather than the headline numbers project 140 supplies.

What You'll Build & Learn

Group crawled URLs under their 2xx, 3xx, 4xx, and 5xx family classes to form a report body.

- Reports list URLs, not just counts
- defaultdict avoids missing-key checks
- Family label keys the groups

🧭 Skill Roadmap

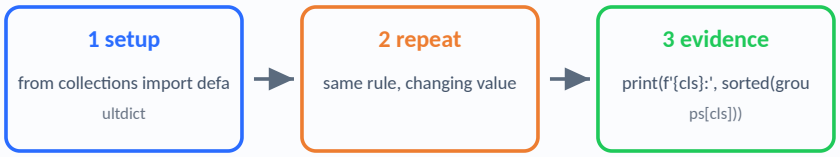
Skill step: pushes the same idea into a less automatic situation. It builds on Project 144: validate that canonicals point at live pages. Project 146 changes the surface problem while preserving the underlying method. *progress check*

Reading target: group crawled urls by status class is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{cls}:', sorted(groups[cls]))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`for url, code in crawl:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{cls}:', sorted(groups[cls]))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for url, code in crawl:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Group Crawled URLs by Status Class: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: collections, defaultdict, crawl, https, groups. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: for url, code in crawl: | Evidence: print(f'{cls}:', sorted(groups[cls]))

🛠️ Reader to Writer Bridge

1. Read: explain group crawled urls by status class in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 144.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 146 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{cls}:', sorted(groups[cls]))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of group crawled urls by status class and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `defaultdict(list)` keyed by the `Nxx` class collects URLs into buckets in one pass; sorting both the classes and each bucket keeps the report stable.

```
from collections import defaultdict
crawl = [
    ('https://a/', 200), ('https://b/', 301),
    ('https://c/', 404), ('https://d/', 500),
    ('https://e/', 200), ('https://f/', 404),
]
groups = defaultdict(list)
for url, code in crawl:
    groups[f'{code // 100}xx'].append(url)
for cls in sorted(groups):
    print(f'{cls}:', sorted(groups[cls]))
```

```
output
2xx: ['https://a/', 'https://e/']
3xx: ['https://b/']
4xx: ['https://c/', 'https://f/']
5xx: ['https://d/']
```

How it works:

1. crawl is a list of URL and code tuples spanning all four families.
2. groups is a defaultdict of lists keyed by family label.
3. The loop appends each URL to the bucket named by its status family.
4. It then prints each sorted family label followed by its sorted URLs.

Try it:

1. Sort each family's URLs by some priority instead of alphabetically.
2. Print only the families that contain error URLs.
3. Add the per-family count from project 140 above each list.
4. Print only the 4xx and 5xx buckets (the problems).

⚠ Common Mistakes

- Appending to a plain dict without a default raises KeyError on the first URL of a class.
- Forgetting to sort each bucket's URLs makes the listing order vary.
- Grouping by exact code instead of class produces too many tiny buckets for a summary.

Skills gained `URL grouping` `collections.defaultdict` `family labels`

→ **Where You'll Use This:** The grouped listing forms the detailed body of a crawl report, letting an SEO click into the actual 404 and 5xx URLs behind each family count to plan fixes.

Mental Model

Like filing every complaint into a labelled folder so you can open the Errors drawer and read each case, not just its tally.

Project 146 Diff Two Crawls for Status Regressions

The Idea Behind It

Comparing a fresh crawl against the last one surfaces pages that newly broke. This code holds before and after status maps and walks the before URLs: a page that was 200 and is no longer a 2xx is reported as REGRESSED, while any other change prints as a plain change. The status regression is the alert an audit fires on, distinguishing breakage from harmless shifts.

What You'll Build & Learn

Diff two crawls to surface status regressions where a page that was 200 now returns an error.

- Diffing crawls catches new breakage
- A regression is 200 turned error
- Distinguishing regressions from changes

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 145: group crawled urls by status class. Project 147 changes the surface problem while preserving the underlying method.

progress check

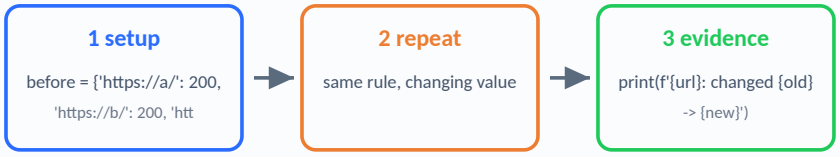
Reading target: diff two crawls for status regressions is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `before = {'https://a/': 200, 'https://b/': 200, 'https://c/': 301}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.

- Evidence: `print(f'{url}: changed {old} -> {new}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`before = {'https://a/': 200, 'https://b/': 200, 'https://c/': 301}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if old == 200 and not (200 <= new < 300):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{url}: changed {old} -> {new}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if old == 200 and not (200 <= new < 300):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Diff Two Crawls for Status Regressions: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: before, https, after, url, sorted. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: before = {'https://a/': 200, 'https://b/': 200, 'https://c/': 301} | Core: if old == 200 and not (200 <= new < 300): | Evidence: print(f'{url}: changed {old} -> {new}')

🛠️ Reader to Writer Bridge

1. Read: explain diff two crawls for status regressions in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 145.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 147 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{url}: changed {old} -> {new}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `before = {'https://a/': 200, 'https://b/': 200, 'https://c/': 301}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of diff two crawls for status regressions and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

For each URL the prior and current statuses are compared; a page that was 200 and is no longer a success is reported as a regression, other changes are noted.

```
before = {'https://a/': 200, 'https://b/': 200, 'https://c/': 301}
after = {'https://a/': 200, 'https://b/': 404, 'https://c/': 200}
for url in sorted(before):
    old, new = before[url], after.get(url, 0)
    if old == 200 and not (200 <= new < 300):
        print(f'{url}: REGRESSED {old} -> {new}')
    elif old != new:
        print(f'{url}: changed {old} -> {new}')
```

```
output
https://b/: REGRESSED 200 -> 404
https://c/: changed 301 -> 200
```

How it works:

1. before and after are status maps for the same three URLs.
2. The loop reads old from before and new from after, defaulting to 0 if absent.
3. When old was 200 and new is no longer a 2xx it prints REGRESSED.
4. Any other differing pair prints as a plain change.

Try it:

1. Also report URLs present in after but missing from before.
2. Flag a recovery where an error in before is now 200.
3. Group regressions by their new status class in the output.
4. Report URLs that recovered (error before, 200 now).

⚠ Common Mistakes

- Comparing only exact codes flags a harmless 200-to-200 as unchanged but misses class-level regressions.
- Using `after[url]` raises KeyError for a URL dropped from the new crawl.
- Treating a 200-to-301 as a regression may be wrong if the redirect is intentional.

Skills gained crawl diff | regression detection | default lookup

→ Where You'll Use This: Scheduled crawls diff against the prior run so an SEO is alerted the moment a working page breaks, separating genuine regressions from benign status changes like a redirect resolving.

Mental Model

Like comparing today's inventory against yesterday's to spot which shelves emptied overnight.

Project 147 Resolve a Chain but Stop at a Hop Limit

The Idea Behind It

Browsers and crawlers cap redirect depth and abort overly long chains. This walker follows the 301 and 302 hops but stops once a configurable limit is reached, even if more redirects remain. With a limit of two on a three-hop chain, the loop halts at the third URL still redirecting, so the code reports it ABORTED. Modelling a hard hop limit mirrors that real give-up behaviour.

What You'll Build & Learn

Resolve a redirect chain but abort once a configured hop limit is reached, like a real client.

- Clients cap redirect depth
- Combining status and hop guards
- Reporting an abort versus a resolution

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 146: diff two crawls for status regressions. Project 148 changes the surface problem while preserving the underlying method.

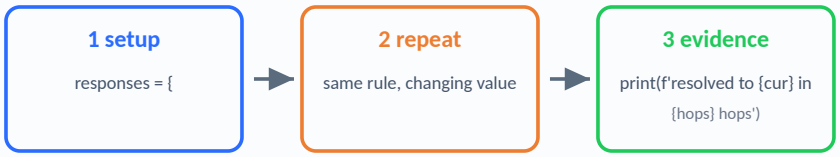
progress check

Reading target: resolve a chain but stop at a hop limit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'resolved to {cur} in {hops} hops')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if responses[cur]['status'] in (301, 302):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'resolved to {cur} in {hops} hops')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if responses[cur]['status'] in (301, 302):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Resolve a Chain but Stop at a Hop Limit: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, limit. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = { | Core: if responses[cur]['status'] in (301, 302): | Evidence: print(f'resolved to {cur} in {hops} hops')

🛠️ Reader to Writer Bridge

1. Read: explain resolve a chain but stop at a hop limit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 146.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 148 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'resolved to {cur} in {hops} hops')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of resolve a chain but stop at a hop limit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

The walk stops either at a non-redirect page or when the hop limit is hit; checking the final status distinguishes a clean resolution from a forced abort.

```
responses = {
    'https://1/': {'status': 301, 'location': 'https://2/'},
    'https://2/': {'status': 301, 'location': 'https://3/'},
    'https://3/': {'status': 301, 'location': 'https://4/'},
    'https://4/': {'status': 200, 'location': None},
}
```

```
limit, cur, hops = 2, 'https://1/', 0
while responses[cur]['status'] in (301, 302) and hops < limit:
    cur = responses[cur]['location']
    hops += 1
if responses[cur]['status'] in (301, 302):
    print(f'ABORTED after {hops} hops at {cur}')
else:
    print(f'resolved to {cur} in {hops} hops')
```

```
output
ABORTED after 2 hops at https://3/
```

How it works:

1. The responses dictionary defines a three-hop chain ending in 200.
2. limit, cur, and hops are initialised together with limit at 2.
3. The while loop advances only while the status is a redirect and hops is below limit.
4. If the stopping URL is still a redirect it prints ABORTED, otherwise it reports the resolution.

Try it:

1. Make the limit a function argument and try several values.
2. Record the partial chain walked before the abort.
3. Match a specific client's real redirect cap, such as twenty hops.
4. Raise `limit` to 3 and watch the chain resolve.

⚠ Common Mistakes

- Reporting success whenever the loop exits ignores that it may have exited on the hop cap.
- Setting the limit one too low aborts chains that would have resolved on the next hop.
- Counting hops after the limit check lets the walk overshoot by one.

Skills gained hop limit | compound loop guard | abort reporting

→ Where You'll Use This: Crawlers and browsers enforce a maximum redirect depth; modelling it lets an audit predict which chains a search engine will abandon rather than follow to their end.

⚡ Mental Model

Like a courier with a rule that after two forwarding stops they give up and return the parcel undelivered.

Project 148 Trace and Report a Multi-Step Redirect Loop

💡 The Idea Behind It

A loop can span several hops before closing, and reporting the exact looping segment helps an engineer fix it. This walker carries the visited path as a list; when it meets a URL already in the path, it slices from that URL's first appearance to the end and prints the looping segment. A while-else clause prints the destination when no loop is found, so the trace shows precisely where the cycle starts.

What You'll Build & Learn

Trace a multi-hop redirect loop and print the exact segment of URLs that forms the cycle.

- Loops can span several hops
- A path list locates the cycle start
- while-else runs when no loop occurs

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 147: resolve a chain but stop at a hop limit. Project 149 changes the surface problem while preserving the underlying method.

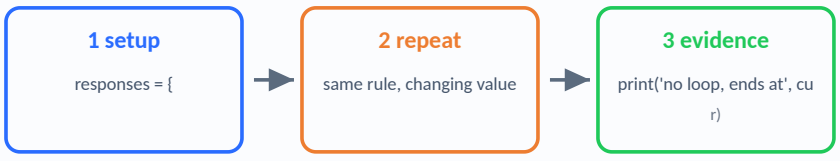
progress check

Reading target: trace and report a multi-step redirect loop is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('no loop, ends at', cur)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if cur in path:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('no loop, ends at', cur)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if cur in path:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Trace and Report a Multi-Step Redirect Loop: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, cur. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: responses = { | Core: if cur in path: | Evidence: print('no loop, ends at', cur)

🛠️ Reader to Writer Bridge

1. Read: explain trace and report a multi-step redirect loop in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 147.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 149 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('no loop, ends at', cur)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of trace and report a multi-step redirect loop and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Tracking the ordered path of visited URLs lets the code slice out the exact cycle (`b -> c -> b`) the moment a URL recurs, rather than just reporting a loop exists.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://b/'},
    'https://b/': {'status': 301, 'location': 'https://c/'},
    'https://c/': {'status': 302, 'location': 'https://b/'},
}
cur, path = 'https://a/', []
while responses[cur]['status'] in (301, 302):
    if cur in path:
        seg = path[path.index(cur):] + [cur]
        print('loop:', ' -> '.join(seg))
        break
    path.append(cur)
    cur = responses[cur]['location']
else:
    print('no loop, ends at', cur)
```

```
output
loop: https://b/ -> https://c/ -> https://b/
```

How it works:

1. The responses dictionary chains three redirects whose last hop loops back to the middle URL.
2. cur starts at the first URL and path is an empty list of visited URLs.
3. When cur is already in path, the code slices from its first index to build the looping segment and prints it.
4. The while-else clause prints no loop and the destination if the loop never triggers.

Try it:

1. Also print the prefix of URLs that led into the loop.
2. Add a hop cap as a backstop in case the data has no entry for a URL.
3. Report the length of the looping segment as a number.
4. Break the loop and confirm the else branch prints 'no loop'.

⚠ Common Mistakes

- Using a set instead of an ordered list loses the ability to print the looping segment in order.
- Slicing from the wrong index would include pre-loop URLs in the reported cycle.
- The `while/else` only runs the else branch if the loop ends normally; misplacing it would misreport.

Skills gained loop reporting | path slicing | while-else

→ Where You'll Use This: Reporting the precise cycle segment turns a vague redirect loop alert into an actionable fix, showing an engineer exactly which URLs to break the cycle between.

⚡ Mental Model

Like retracing your steps through a building and marking the exact corridor where the path circles back on itself.

Project 149 Score the Health of a Redirect Audit

💡 The Idea Behind It

Rolling several redirect checks into one score lets you track redirect hygiene over time. This code holds a findings count per defect type and a weights dictionary assigning each defect a severity. It multiplies counts by weights and sums them into a penalty, then subtracts that from a clean baseline of 100, clamped at zero. The output shows the penalty and the resulting health score.

What You'll Build & Learn

Combine weighted redirect-audit findings into a single health score out of 100.

- Weighted defects roll into one score
- Severity weights rank defect types

- max clamps the score at zero

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 148: trace and report a multi-step redirect loop. Project 150 changes the surface problem while preserving the underlying method.

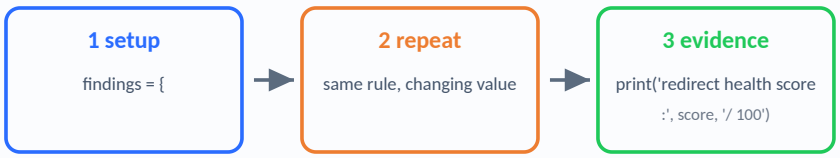
progress check

Reading target: score the health of a redirect audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `findings = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print('redirect health score:', score, '/ 100')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`findings = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`penalty = sum(findings[k] * weights[k] for k in findings)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print('redirect health score:', score, '/ 100')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `penalty = sum(findings[k] * weights[k] for k in findings)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score the Health of a Redirect Audit: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: findings, loops, over_limit_chains, ends_in_4xx, temp_should_be_perm. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: findings = { | Core: penalty = sum(findings[k] * weights[k] for k in findings) | Evidence: print('redirect health score:', score, '/ 100')

Reader to Writer Bridge

1. Read: explain score the health of a redirect audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 148.
   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 150 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('redirect health score:', score, '/ 100')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `findings = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score the health of a redirect audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each defect type is weighted by severity, multiplied by its count, summed into a penalty, and subtracted from 100 with a floor at zero to give a single health score.

```
findings = {
    'loops': 1,
    'over_limit_chains': 2,
    'ends_in_4xx': 1,
    'temp_should_be_perm': 3,
}
weights = {'loops': 10, 'over_limit_chains': 3, 'ends_in_4xx': 5,
'temp_should_be_perm': 2}
penalty = sum(findings[k] * weights[k] for k in findings)
score = max(0, 100 - penalty)
print('penalty:', penalty)
print('redirect health score:', score, '/ 100')
```

```
output
penalty: 27
redirect health score: 73 / 100
```

How it works:

1. findings counts each defect type, like loops and over-limit chains.
2. weights assigns a severity to each defect type.
3. penalty sums findings times weights across the shared keys.
4. score subtracts penalty from 100 and clamps at zero with max, then both are printed.

Try it:

1. Add a new defect type with its own weight and count.

2. Cap each defect's contribution so one category cannot dominate.
3. Convert the score into a letter grade for the report.
4. Set every finding to zero and confirm a perfect 100.

⚠ Common Mistakes

- Letting the score go negative without a `max(0, ...)` floor produces a meaningless number.
- Weighting a minor defect as heavily as a loop hides which problems matter most.
- Summing counts without weights treats every defect as equally severe.

Skills gained health scoring | weighted sum | max clamp

→ Where You'll Use This: A single redirect health score lets teams track hygiene across releases and set a quality gate; weighting defects by severity ensures a loop hurts the score more than an extra hop.

Mental Model

Like a credit score that starts clean and deducts points weighted by how serious each blemish is.

Project 150 Run a Full Status and Redirect Audit

The Idea Behind It

This capstone ties the chapter together over one response map. The resolve function walks each URL's chain with a seen set, returning the looping URL when a cycle is found or otherwise the final URL and status. Looping over the sorted map, the code resolves every URL and prints its destination with a status class or a loop marker. The sample surfaces a clean 200, a 404 dead-end, and a self-loop in a single pass.

What You'll Build & Learn

Run a combined audit that resolves every URL's redirect chain and reports its final status or loop.

- A resolver unifies the chapter's checks
- seen set handles loops inside resolve
- One pass reports status and breakage

Skill Roadmap

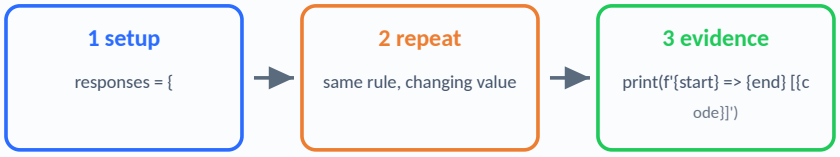
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 149: score the health of a redirect audit. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run a full status and redirect audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `responses = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
- Evidence: `print(f'{start} => {end} [{code}]')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
2. Apply the rule	`if cur in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{start} => {end} [{code}]')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: HTTP Status Codes and Redirects becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if cur in seen:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Run a Full Status and Redirect Audit: connect the surface task to the chapter skill, classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.
Names	Watch these names first: responses, https, status, location, resolve. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: responses = { | Core: if cur in seen: | Evidence: print(f'{start} => {end} [{code}]')

🛠️ Reader to Writer Bridge

1. Read: explain run a full status and redirect audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 149.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{start} => {end} [{code}]')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `responses = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run a full status and redirect audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

One resolver walks each URL's redirects with a seen-set guard, returning either the terminal status or a loop marker, and the loop over sorted starts produces the full audit line by line.

```
responses = {
    'https://a/': {'status': 301, 'location': 'https://b/'},
    'https://b/': {'status': 200, 'location': None},
    'https://c/': {'status': 301, 'location': 'https://d/'},
    'https://d/': {'status': 404, 'location': None},
    'https://e/': {'status': 301, 'location': 'https://e/'},
}
def resolve(start):
    cur, seen = start, set()
    while responses[cur]['status'] in (301, 302):
        if cur in seen:
            return cur, 'loop'
        seen.add(cur)
        cur = responses[cur]['location']
    return cur, responses[cur]['status']
for start in sorted(responses):
    end, code = resolve(start)
    print(f'{start} => {end} [{code}]')
```

```
output
https://a/ => https://b/ [200]
https://b/ => https://b/ [200]
https://c/ => https://d/ [404]
https://d/ => https://d/ [404]
https://e/ => https://e/ [loop]
```

How it works:

1. The responses dictionary holds a clean chain, a chain ending in 404, and a self-redirect.
2. resolve walks from a start URL, returning loop if it revisits a URL in seen.
3. Otherwise it returns the final URL and its status code.
4. The loop resolves every sorted URL and prints its destination with the status or loop marker.

Try it:

1. Add a hop limit so an unbounded chain aborts cleanly.
2. Roll the per-URL outcomes into the family summary from project 140.

3. Flag chains that end in 4xx or 5xx separately from clean ones.
4. Add a chain ending in 500 and watch it surface as a server error.

⚠ Common Mistakes

- Resolving without the seen-set guard makes the self-redirect on `https://e/` loop forever.
- Reporting only the start URL's status ignores where the chain actually lands.
- Iterating an unsorted map makes the audit output order nondeterministic.

Skills gained full audit | chain resolution | loop handling

→ Where You'll Use This: This resolver is the skeleton of a real redirect audit; pointed at live crawl data it classifies every URL's outcome at once, the engine behind a status-and-redirect report.

Mental Model

Like a single inspection pass that walks every forwarding trail and stamps each one delivered, dead-end, or stuck in a loop.

✅ Chapter Progress Checkpoint

What this chapter added: after 40 mini-projects, HTTP Status Codes and Redirects should feel like a reusable Python move: classify 2xx/3xx/4xx/5xx, trace redirect chains and loops, and tell 301 from 302 from canonical.

chapter checkpoint

1 Read
Open any project from Projects 111-150 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 5 (On-Page Signals and the HTML Audit) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

8 questions · 4 categories · graded

2 Recall | 2 Meaning | 2 Structure | 2 Transfer

4.1.Q1 Recall MEDIUM Turn back to Project 118 (Detect a Redirect That Dead-Ends in 404). Without running it, what exactly does it print?

4.1.Q2 Meaning MEDIUM Turn back to Project 143 (Spot a Redirect That Only Strips Query Params). Without running it, what exactly does it print?

4.1.Q3 Structure MEDIUM Turn back to Project 123 (Compute the 5xx Error Rate of a Crawl). Without running it, what exactly does it print?

4.1.Q4 Transfer MEDIUM Turn back to Project 113 (Walk a Redirect Chain to the Final URL). Without running it, what exactly does it print?

4.1.Q5 Recall EASY A server returns the code 503. Which class is it, and what should a crawler do?

- A 2xx success — index it
- B 3xx redirect — follow Location
- C 4xx client error — drop the URL
- D 5xx server error — retry later

4.1.Q6 Meaning MEDIUM You permanently move /old-page to /new-page but the redirect returns a 302. Why is that a problem for SEO?

- A 302 is an error code
- B A 302 is temporary, so engines keep the old URL indexed and do not fully transfer ranking signals to the new URL
- C 302 redirects are slower than 301s
- D Browsers ignore 302 redirects

4.1.Q7 Structure MEDIUM A page returns status 200, but its body reads "Sorry, that page no longer exists." What is this called and why does it hurt?

4.1.Q8 Transfer MEDIUM When should you use a canonical tag instead of a 301 redirect to consolidate two duplicate URLs?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 4 – HTTP Status Codes and Redirects

4.1.Q1. `final URL: https://gone/ status 404` then `BROKEN redirect`

4.1.Q2. `https://s/p?ref=ad -> https://s/p: query-strip` then `https://s/a -> https://s/b: page-move`

4.1.Q3. `3/8 server errors = 37.5%`

4.1.Q4. `https://a/ -> https://b/ -> https://c/` then `final: https://c/ status 200`

4.1.Q5. D 503 (Service Unavailable) is a 5xx server error; it is transient, so a well-behaved crawler retries later rather than dropping the page.

4.1.Q6. B a permanent move needs a 301 (or 308); a 302 tells engines the move is temporary, so link equity stays stranded on the old URL.

4.1.Q7. A soft-404: the 200 code says the page is fine while the body admits it is missing, so engines waste crawl budget indexing an empty page. The fix is to return a real 404 or 410.

4.1.Q8. When users still need the duplicate URL to keep working (e.g. a tracking-parameter variant like?ref=ad). A canonical keeps both URLs live but tells engines which to index; a 301 removes the duplicate entirely.



PART II

Intermediate Skills

With the basics in hand, take on functions, data structures, recursion, algorithms and object-oriented design.

Chapter 5 – On-Page Signals and the HTML Audit · Chapter 6 – Keyword Research and Intent as Data · Chapter 7 – Canonicalization, Duplicate and Thin Content · Chapter 8 – Site Speed and Core Web Vitals · Chapter 9 – Structured Data and Rich Results

Chapter 5 · On-Page Signals and the HTML Audit

Audit titles, meta descriptions, header hierarchy, alt text and intent match from an HTML snippet.

Chapter Skill Map

What this chapter adds: In On-Page Signals and the HTML Audit, the focus is audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet. You practise it across 40 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 5 assumes the reading habits from Chapter 4: HTTP Status Codes and Redirects. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 6, Keyword Research and Intent as Data, reuses this chapter's discipline with a new kind of problem.

1 **Projects 151-163**	2 **Projects 164-176**	3 **Projects 177-190**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

On-page SEO lives in a handful of HTML tags. An audit is just reading those tags and asking, for each, "is this what a search engine wants to see?" *the page tells you, if you parse it*

Once the crawler reaches a page and indexes it, ranking is shaped by *on-page signals* — facts the engine reads directly out of your own HTML. The good news for an auditor: every one of these signals is a tag or attribute you can find with a few lines of code. The `title` is one tag. The `meta description` is one tag. The `H1` is one tag. The `canonical` and `robots` directives are link and meta tags in the `<head>`. There is no guessing — you read the markup and compare it to a rule.

Each signal has a *known healthy range*. A title should be roughly 50-60 characters so the search-results page does not truncate it. A meta description should stay under ~155 characters for the same reason. A page should carry `exactly one` `<h1>` — not zero, not three. Headings should step down one level at a time (h1 → h2 → h3), never skip. Every meaningful image needs `alt` text. The whole craft of an HTML audit is encoding these ranges as checks.

The simplest signal is also the strongest: the `<title>`. Reading it is one regular expression, and its length decides whether it survives in the results snippet. Here is the first audit in the chapter — pull the title out of the head and judge its length against the 50-60 sweet spot:

```
import re
html = '<head><title>Best Running Shoes for Marathon Training in 2026</title>
</head>'
m = re.search(r'<title>(.*?)</title>', html, re.S)
title = m.group(1).strip() if m else ''
```

```
n = len(title)
verdict = 'OK' if 50 <= n <= 60 else ('too short' if n < 50 else 'too long')
print(f'title ({n} chars): {verdict}')
print(title)
```

```
output
title (48 chars): too short
Best Running Shoes for Marathon Training in 2026
```

That tiny program is the shape of the entire chapter: extract the signal, measure it, compare it to a rule, print a verdict. Swap the regex and the rule and you have a description-length check, an H1 counter, an alt-text scanner, or a canonical detector. Stack a dozen of them together and you have a real on-page audit report.

extract → measure → judge

One caution runs through the whole chapter: regex is a quick way to read HTML but a brittle one. It does not decode entities like `&`, and it gets confused by nested tags. For robust extraction the standard library ships `html.parser.HTMLParser`, which we reach for when the markup gets real. Most checks here use regex for clarity, but you should know its limits.

TITLE TAG

The `<title>` in the document head. It supplies the clickable headline in search results and is the single strongest on-page relevance signal. Search engines truncate it around 60 characters.

META DESCRIPTION

A `<meta name="description" content="...">` tag whose text often becomes the snippet under the title in results. Not a direct ranking factor, but it drives click-through; clipped near ~155 characters.

HEADING HIERARCHY

The ordered set of `<h1>`–`<h6>` tags. A well-formed page has exactly one H1 (its main topic) and steps levels down one at a time, forming a readable outline.

ALT TEXT

The `alt` attribute on an `<img>`. It describes the image for screen readers and gives crawlers text to index. A *missing* alt is a fault; an intentionally empty `alt=""` marks a decorative image to be skipped.

CANONICAL LINK

A `<link rel="canonical" href="...">` naming the master URL for a page, so duplicate or parameter variants consolidate their ranking signals onto one address instead of competing.

ROBOTS META

A `<meta name="robots" content="...">` carrying crawl/index directives. `noindex` keeps the page out of the index; `nofollow` tells engines not to pass ranking credit through its links.

Title and description: the SERP snippet, measured

The title and meta description are what a searcher actually *sees* in the results — the headline and the blurb. Both get cut off if they run long, so length is the first thing an audit checks. The title's healthy band is 50-60 characters; the description should stay under ~155. Reading the description is the same one-regex move as the title, just a different tag:

```
import re
html = '<meta name="description" content="Shop our full range of trail and road
running shoes with free returns and expert fitting advice for every runner.">'
m = re.search(r'<meta\s+name="description"\s+content="([^"]*)"', html)
desc = m.group(1) if m else ''
n = len(desc)
verdict = 'OK' if n <= 155 else 'too long'
print(f'description ({n} chars): {verdict}')
```

```
output
description (113 chars): OK
```

Length is a proxy, not the truth — SERPs truncate by *pixel width*, not character count, so a short count of wide words can still be clipped. A rough estimate of ~9.5 pixels per character flags titles likely to be cut, catching the wide-word case a character count misses.

characters are a guess; pixels are the rule

The heading outline: one H1, no skipped levels

Headings are the page's table of contents. Two rules matter. First, a page should carry exactly one `<h1>` naming its main topic — zero leaves the page topic-less, several make competing claims. Counting H1s catches both faults at once:

```
import re
html = '<body><h1>Running Shoes</h1><p>Intro</p><h2>Trail</h2><h1>Sale</h1></body>'
h1s = re.findall(r'<h1[^>]*>(.*?)</h1>', html, re.S)
n = len(h1s)
verdict = 'OK' if n == 1 else ('missing H1' if n == 0 else 'multiple H1')
```

```
print(f'h1 count = {n}: {verdict}')
print('h1 texts:', [t.strip() for t in h1s])
```

```
output
h1 count = 2: multiple H1
h1 texts: ['Running Shoes', 'Sale']
```

Second, levels should step down one at a time. Jumping h2 → h4 *skips* a level, which screen readers and crawlers read as a broken outline. Collect the level numbers in document order and look at each adjacent pair: a jump of more than one is a skip.

```
import re
html = '<h1>Guide</h1><h2>Setup</h2><h4>Details</h4><h2>Usage</h2><h3>Tips</h3>'
levels = [int(t) for t in re.findall(r'<h([1-6])[^>]*>', html)]
issues = []
for prev, cur in zip(levels, levels[1:]):
    if cur > prev + 1:
        issues.append(f'h{prev} -> h{cur}')
print('levels:', levels)
print('skips:', issues if issues else 'none')
```

```
output
levels: [1, 2, 4, 2, 3]
skips: ['h2 -> h4']
```

Render the same headings as an indented tree and the structure becomes visible at a glance — that is just printing each heading with two spaces of indent per level above one. A good audit shows the outline *and* names the faults.

Images: missing alt versus deliberately empty

An `<img>` with no `alt` attribute is invisible to screen readers and gives crawlers no text to index. Scanning every image tag for a missing alt is a core check. But there is a subtlety the audit must respect: a *decorative* image (a spacer, a divider) should carry `alt=""` so assistive tech skips it. So three states exist — `missing` (a fault), `empty/decorative` (fine), and `described` (ideal):

```
import re
html = '<img src="a.jpg" alt="Product photo"><img src="spacer.gif" alt=""><img
src="b.png" alt="   ">'
for tag in re.findall(r'<img\b[^>]*>', html):
    m = re.search(r'\balt\s*=\s*"([^"]*)"', tag)
    src = re.search(r'src\s*=\s*"([^"]*)"', tag).group(1)
    if m is None:
        kind = 'MISSING alt'
    elif m.group(1).strip() == '':
        kind = 'empty alt (decorative)'
    else:
```

```
        kind = 'described'
    print(f'{src}: {kind}')
```

```
output
a.jpg: described
spacer.gif: empty alt (decorative)
b.png: empty alt (decorative)
```

Note that `alt=" "` (only whitespace) collapses to empty after `.strip()` — it counts as decorative, not described. Reusing one alt string across many images is its own miss: helpful to nobody, so an audit also tallies alt values to flag copy-pasted descriptions.

Head directives: canonical and robots

Two tags in the `<head>` quietly decide a page's fate in the index. The `canonical` link names the master URL so duplicate variants do not dilute each other. Confirm it exists and read where it points:

```
import re
html = '<head><link rel="stylesheet" href="/s.css"><link rel="canonical"
href="https://example.com/blog/post"></head>'
m = re.search(r'<link\b[^>]*rel\s*=\s*"canonical"[^>]*>', html)
if m:
    href = re.search(r'href\s*=\s*"([^"]*)"', m.group(0)).group(1)
    print('canonical present:', href)
else:
    print('canonical MISSING')
```

```
output
canonical present: https://example.com/blog/post
```

The `robots` meta is more dangerous when wrong: a stray `noindex` silently removes a page from search entirely, and `nofollow` stops its links passing credit. The directives are comma-separated, so split them and test for the two killers:

```
import re
html = '<meta name="robots" content="index, follow, max-snippet:-1">'
m = re.search(r'<meta\s+name="robots"\s+content="([^"]*)"', html)
directives = [d.strip().lower() for d in m.group(1).split(',')] if m else []
indexable = 'noindex' not in directives
followable = 'nofollow' not in directives
print('directives:', directives)
print('indexable:', indexable, '| followable:', followable)
```

```
output
directives: ['index', 'follow', 'max-snippet:-1']
indexable: True | followable: True
```

Keyword placement and intent match

Whether a keyword appears matters; *where* it appears matters as much. A target term should surface in the title, the H1, and ideally the opening paragraph. Checking all three at once shows whether the page's most-read elements actually mention the topic:

```
import re
html = '<title>Trail Running Shoes Review</title><h1>Best Trail Running Shoes</h1>
<p>Our guide covers grip, cushioning and fit for every runner.</p>'
keyword = 'trail running shoes'
title = (re.search(r'<title>(.*?)</title>', html, re.S) or [None, ''])[1].lower()
h1 = (re.search(r'<h1[^>]*>(.*?)</h1>', html, re.S) or [None, ''])[1].lower()
first_p = (re.search(r'<p[^>]*>(.*?)</p>', html, re.S) or [None, ''])[1].lower()
for name, hay in [('title', title), ('h1', h1), ('first paragraph', first_p)]:
    print(f'{name}: {"yes" if keyword in hay else "no"}')
```

```
output
title: yes
h1: yes
first paragraph: no
```

Placement should also fit *intent*. A title leaks the searcher it targets: "how to" and "guide" signal informational intent, "buy" and "price" signal transactional. If a product page's title reads informational, the content and the target are mismatched. Scoring intent cue words in the title classifies it, and comparing that to the page's intended intent flags the clash. Restraint matters too — a description that repeats "best" five times reads as keyword stuffing to engine and reader alike.

right word, right place, right intent

Example 1

A page has `<title>Buy Cheap Running Shoes Online Now Free Shipping Best Deals 2026 Sale</title>` (76 characters). Its character count is over 60, but is that the only reason it will be clipped? Judge it by pixel width too (~9.5px per character; SERPs cut around 600px).

SOLUTION

1. By character count: 76 chars is already over the 50-60 band, so it is too long.
2. Estimate pixel width: 76 × 9.5 ≈ 722px, well past the ~600px the results page shows.
3. Both measures agree it gets clipped — but the pixel estimate is the one that catches wide-word titles a 60-char rule would pass.

Clipped on both counts: 76 chars and ~722px both exceed their limits. The fix is a shorter, front-loaded title.

pixel width is why even a 58-char title of wide words can truncate

Example 2

Audit this `<head>` against four essentials — non-empty title, a description, a canonical, and a viewport: `<head><title>Pricing Plans</title><meta name="description" content="Compare our plans."></head>`. What is the score?

SOLUTION

1. Title: `Pricing Plans` is present and non-empty → pass.
2. Description: the meta description tag is present with content → pass.
3. Canonical: no `<link rel="canonical">` in the head → MISSING.
4. Viewport: no `<meta name="viewport">` → MISSING. Two of four pass.

Score 2/4 — title and description present, canonical and viewport missing. A real audit rolls many such checks into one scorecard. *one HTML string in, one pass/fail tally out*

Example 3

The page targets transactional intent (it is a product page). Its title is `How to Choose Running Shoes: Buying Guide`. Does the title's intent match the page's target?

SOLUTION

1. Score intent cue words in the lowercased title. Informational cues: "how", "guide" → 2 hits. Transactional cues: "buy", "shop", "price" → none of those exact words appear ("buying" is not "buy") → 0 hits.
2. The strongest signal is informational, so the title *reads* informational.
3. But the page is built to be transactional, so detected (informational) ≠ target (transactional).

NO — intent mismatch. The title signals informational while the page targets transactional, a content-versus-target clash worth fixing. *a product page whose title says "how to" is fighting itself*

⚡ Mental-Math Shortcut

Extract, measure, judge. Every on-page check is the same three beats: pull the signal out of the HTML, measure it (a length, a count, a presence), and compare it to a known-healthy rule. If you can name the tag and the rule, you can write the check — and a stack of those checks *is* the audit. When in doubt about the rule of thumb: title 50-60 chars, description under ~155, exactly one H1, no skipped heading levels, every meaningful image alt-described, canonical and viewport present, no stray noindex.

⚠ Common Mistakes

- Trusting a character count for title length when SERPs truncate by pixel width — a short count of wide words still gets clipped, so estimate pixels too.
- Treating an empty `alt=""` as a fault. A *missing* alt is the problem; an intentionally empty alt correctly marks a decorative image for screen readers to skip.

- Reading HTML with regex when entities or nested tags are in play — `&` stays raw and nested markup confuses the pattern. Reach for `html.parser.HTMLParser` when the markup is real.
- Optimising whether a keyword is present while ignoring where and with what intent. A keyword buried at the end of an informational title on a transactional page is a weak, mismatched signal.

This chapter contains 40 hands-on mini-projects (Projects 151–190). Type each one out, run it, and experiment.

Project 151 Check the Title Tag Length

The Idea Behind It

The title tag is the loudest on-page relevance signal you control, and search engines clip it near sixty characters. This project pulls the text between the title tags with a non-greedy regex, strips whitespace, measures its character length, and labels it too short, too long, or inside the fifty-to-sixty sweet spot.

What You'll Build & Learn

Read a page title from raw HTML and judge its length against the range that displays cleanly in a result snippet.

- Why title length matters for snippets
- Non-greedy regex capture with DOTALL
- Three-way verdict from a length comparison

Skill Roadmap

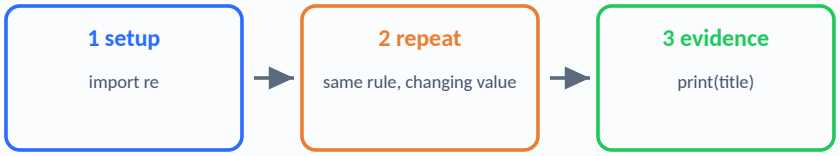
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 152 changes the surface problem while preserving the underlying method. *progress check*

Reading target: check the title tag length is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(title)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`title = m.group(1).strip() if m else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(title)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `title = m.group(1).strip() if m else ''`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check the Title Tag Length: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, head, title, Best. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: title = m.group(1).strip() if m else '' | Evidence: print(title)

🛠️ Reader to Writer Bridge

1. Read: explain check the title tag length in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 152 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(title)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check the title tag length and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A non-greedy `(.*?)` between `<title>` and `</title>` captures the title text; its length is bucketed against the 50-60 character window that fits before a SERP truncation.

```
import re
html = '<head><title>Best Running Shoes for Marathon Training in 2026</title>
</head>'
m = re.search(r'<title>(.*?)</title>', html, re.S)
title = m.group(1).strip() if m else ''
n = len(title)
verdict = 'OK' if 50 <= n <= 60 else ('too short' if n < 50 else 'too long')
```

```
print(f'title ({n} chars): {verdict}')
print(title)
```

```
output
title (48 chars): too short
Best Running Shoes for Marathon Training in 2026
```

How it works:

1. An HTML string holds a head with one title element containing a marketing-style page name.
2. re.search with the pattern title group and re.S captures whatever sits between the tags, even across line breaks.
3. The match group is stripped, len gives its character count, and a conditional expression maps that count to a verdict string.
4. The script prints the count with its verdict, then the title text itself on a second line.

Try it:

1. Add a verdict for a completely empty title once whitespace is stripped away
2. Loop the same check over a dictionary of several pages and print one line each
3. Warn when the brand name pushes the keyword past character forty
4. Lengthen the title past 60 characters and watch the verdict flip to 'too long'.

⚠ Common Mistakes

- A greedy `(.*)` would run past the first `</title>` into later markup if the page repeated the tag.
- Counting before `.strip()` lets stray whitespace inside the tag inflate the length.
- Treating a missing title (no match) as length 0 silently labels it 'too short' instead of 'absent'.

Skills gained regex extraction | length validation | title audit

→ Where You'll Use This: Every site crawl and SEO dashboard reports title-length warnings; this is the check that flags pages whose titles will read awkwardly or get truncated in the search results page.

Mental Model

The title is the headline on a newspaper stand: too long and the rack cuts it off, too short and it fails to sell the story.

Project 152 Measure the Meta Description Length

The Idea Behind It

A meta description is the snippet copy that argues for the click, and Google rarely renders past roughly one hundred fifty-five characters before trimming it. The code matches the description meta tag with a regex

anchored on the name attribute, captures the content value, counts its characters, and warns only when that count exceeds the cutoff.

What You'll Build & Learn

Measure a meta description's character length and flag the ones whose closing call to action will be clipped in the snippet.

- The practical meta description length ceiling
- Attribute-anchored regex for a meta tag
- Single-threshold verdict logic

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 151: check the title tag length. Project 153 changes the surface problem while preserving the underlying method.

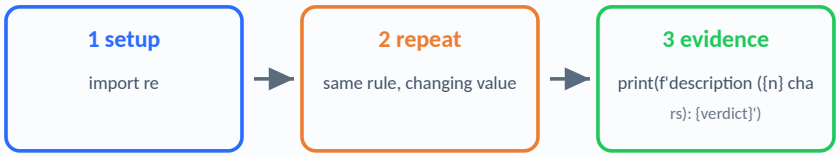
progress check

Reading target: measure the meta description length is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'description ({n} chars): {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`desc = m.group(1) if m else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'description ({n} chars): {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `desc = m.group(1) if m else ''`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure the Meta Description Length: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, meta, name, description. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: desc = m.group(1) if m else '' | Evidence: print(f'description ({n} chars): {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain measure the meta description length in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 151.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 153 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'description ({n} chars): {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure the meta description length and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`[^"]*` captures everything up to the closing quote of the content attribute, and the length is compared against the ~155 character snippet limit.

```
import re
html = '<meta name="description" content="Shop our full range of trail and road running shoes with free returns and expert fitting advice for every runner.">'
m = re.search(r'<meta\s+name="description"\s+content="([^"]*)"', html)
desc = m.group(1) if m else ''
n = len(desc)
verdict = 'OK' if n <= 155 else 'too long'
print(f'description ({n} chars): {verdict}')
```

```
output
description (113 chars): OK
```

How it works:

1. The HTML holds one meta tag whose name is description and whose content is a sentence of shop copy.
2. The regex requires name equals description followed by content, then captures everything inside the content quotes with a negated character class.
3. len measures the captured string and a simple comparison returns OK when at or under one hundred fifty-five, else too long.

4. The print reports the character count alongside the verdict.

Try it:

1. Also warn when the description is shorter than seventy characters and wastes snippet space
2. Handle the case where the description meta tag is entirely absent from the head
3. Switch the attribute order so content precedes name and adjust the pattern
4. Extend the content past 155 characters and watch it flag 'too long'.

⚠ Common Mistakes

- Hard-coding the attribute order (`name` before `content`) misses tags that write `content` first.
- `[^"]*` breaks if the description itself contains an escaped quote entity.
- Treating a missing description as an empty 'OK' hides that the tag is absent entirely.

Skills gained regex matching | meta description | threshold check

→ Where You'll Use This: SEO tools surface truncated descriptions so writers can rewrite snippets that get cut mid-sentence; this length gate is the first thing an audit runs across every indexable page.

⚡ Mental Model

Think of the description as an elevator pitch with a hard time limit: run past the buzzer and the listener only hears half your offer.

Project 153 Count H1 Tags on the Page

💡 The Idea Behind It

A well-formed page carries exactly one H1 naming its main topic, so zero or several is a structural fault. This project uses re.findall to collect the text of every H1 in the markup, counts them, and maps that count to missing, multiple, or OK, then prints the count and the list of captured heading texts.

What You'll Build & Learn

Count the H1 elements on a page and report whether there is none, one, or a problematic surplus.

- The single-H1 rule for page topics
- Collecting all matches with re.findall
- Branching on a count of zero, one, or more

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 152: measure the meta description length. Project 154 changes the surface problem while preserving the underlying method.

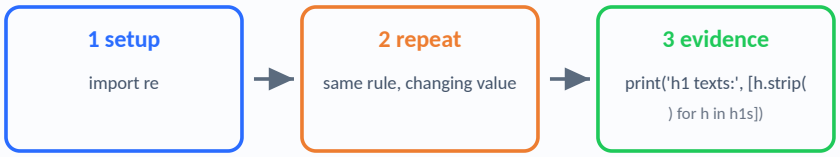
progress check

Reading target: count h1 tags on the page is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('h1 texts:', [h.strip() for h in h1s])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`verdict = 'OK' if n == 1 else ('missing H1' if n == 0 else 'multiple H1')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('h1 texts:', [h.strip() for h in h1s])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'OK' if n == 1 else ('missing H1' if n == 0 else 'multiple H1')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count H1 Tags on the Page: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, body, h1, Running. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: verdict = 'OK' if n == 1 else ('missing H1' if n == 0 else 'multiple H1... | Evidence: print('h1 texts:', [h.strip() for h in h1s])

Reader to Writer Bridge

1. Read: explain count h1 tags on the page in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 152.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 154 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('h1 texts:', [h.strip() for h in h1s])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count h1 tags on the page and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`<h1[^>]*>` tolerates attributes on the tag while `(.*?)` grabs the inner text; the count drives a three-way verdict for missing, single, or duplicate headings.

```
import re
html = '<body><h1>Running Shoes</h1><p>Intro</p><h2>Trail</h2><h1>Sale</h1>
</body>'
h1s = re.findall(r'<h1[^>]*>(.*?)</h1>', html, re.S)
n = len(h1s)
verdict = 'OK' if n == 1 else ('missing H1' if n == 0 else 'multiple H1')
print(f'h1 count = {n}: {verdict}')
print('h1 texts:', [h.strip() for h in h1s])
```

```
output
h1 count = 2: multiple H1
h1 texts: ['Running Shoes', 'Sale']
```

How it works:

1. The body markup deliberately contains two H1 elements plus an H2, simulating a common templating mistake.
2. re.findall with a non-greedy capture and re.S returns a list of every H1's inner text.
3. len of that list is the count, and a nested conditional assigns missing, multiple, or OK.
4. The script prints the count with its verdict and then the stripped texts as a Python list.

Try it:

1. Report which H1 text appears first so an editor knows which to keep
2. Extend the check to also count H2 tags on the same page
3. Treat an H1 that is only whitespace as effectively missing
4. Remove the second `<h1>` and confirm the verdict reads 'OK'.

⚠ Common Mistakes

- `<h1[^>]*>` would also match `<h10>`-style invented tags, though those are not valid HTML.
- A multi-line H1 needs the `re.S` flag or the `.*?` stops at the line break.
- Counting opening tags only (without the closing one) over-counts if a stray `<h1>` is unclosed.

Skills gained findall | heading count | structure audit

→ Where You'll Use This: Crawlers routinely flag pages with multiple H1s left over from old templates or page builders; auditors fix these to give each page one unambiguous topic heading.

Mental Model

An H1 is the title page of a book: one is right, none leaves readers lost, and two staple-bound titles confuse everyone.

Project 154 Detect a Skipped Heading Level

The Idea Behind It

An accessible heading outline steps one level at a time, so jumping from H2 straight to H4 leaves a hole that screen readers and crawlers read as broken structure. The code extracts the numeric level of each heading in order, then zips consecutive pairs and records any jump where the next level exceeds the previous by more than one.

What You'll Build & Learn

Walk the heading levels in document order and report every place where a level was skipped.

- Heading levels should increase by one
- Pairwise iteration with zip and a slice
- Capturing only the digit from a heading tag

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 153: count h1 tags on the page. Project 155 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a skipped heading level is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('skips:', issues if issues else 'none')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if cur > prev + 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('skips:', issues if issues else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if cur > prev + 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Skipped Heading Level: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, h1, Guide, h2. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if cur > prev + 1: | Evidence: print('skips:', issues if issues else 'none')

🛠️ Reader to Writer Bridge

1. Read: explain detect a skipped heading level in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 153.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 155 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('skips:', issues if issues else 'none')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of detect a skipped heading level and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Capturing just the digit in `<h([1-6])>` yields the level sequence, and any adjacent pair where the level jumps by more than one is a skipped-level fault.

```
import re
html = '<h1>Guide</h1><h2>Setup</h2><h4>Details</h4><h2>Usage</h2>
<h3>Tips</h3>'
levels = [int(t) for t in re.findall(r'<h([1-6])[^>]*>', html)]
issues = []
for prev, cur in zip(levels, levels[1:]):
    if cur > prev + 1:
        issues.append(f'h{prev} -> h{cur}')
print('levels:', levels)
print('skips:', issues if issues else 'none')
```

```
output
levels: [1, 2, 4, 2, 3]
skips: ['h2 -> h4']
```

How it works:

1. The HTML has headings at levels one, two, four, two, and three, hiding one skip.
2. re.findall captures just the digit after the h, and a comprehension turns each into an int, giving an ordered levels list.
3. Zipping levels with levels offset by one yields adjacent pairs, and the loop flags any pair where cur exceeds prev plus one.
4. The script prints the levels list and the recorded skips, or none when the outline is clean.

Try it:

1. Also flag when the outline jumps back up more than one level abruptly
2. Record the index position of each skip alongside the level pair
3. Ignore skips inside repeated template sections like a footer if present
4. Insert an `<h3>` before the `<h4>` and watch the skip disappear.

⚠ Common Mistakes

- Only deepening jumps matter; going back up (h4 -> h2) is allowed and must not be flagged.
- `zip(levels, levels[1:])` silently produces nothing for a single-heading page, hiding edge cases.
- Matching `<h[1-6]>` without the digit capture loses the level number you need to compare.

Skills gained zip pairing outline validation regex digits

→ Where You'll Use This: Accessibility and SEO audits check heading order because a jagged outline hurts assistive tech and muddies how engines parse a page's topic hierarchy.

Mental Model

Headings are a staircase: each step should rise one tread, and skipping from the second to the fourth is where someone trips.

Project 155 Build a Heading Outline

The Idea Behind It

The heading hierarchy is effectively a page's table of contents, and rendering it as an indented tree exposes the document's shape at a glance. The code captures each heading's level and inner text together with a backreference, then prints every heading indented by two spaces per level above one, drawing the nesting visually.

What You'll Build & Learn

Turn a page's H1 through H6 tags into a readable indented outline that reveals its structure.

- Visualising structure through indentation
- Backreferences to match opening and closing tags
- Indent width scaled by heading level

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 154: detect a skipped heading level. Project 156 changes the surface problem while preserving the underlying method.

progress check

Reading target: build a heading outline is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print(' ' * (int(lvl) - 1) + text.strip())` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`for lvl, text in heads:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' ' * (int(lvl) - 1) + text.strip())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for lvl, text in heads:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a Heading Outline: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, h1, Coffee, Brewing. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: for lvl, text in heads: | Evidence: print(' ' * (int(lvl) - 1) + text.strip())

Reader to Writer Bridge

1. Read: explain build a heading outline in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 154.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 156 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' ' * (int(lvl) - 1) + text.strip())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a heading outline and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

The back-reference `</h\1>` ties each closing tag to its opening level so heading text is captured cleanly; indenting by `level - 1` reconstructs the nesting.

```
import re
html = '<h1>Coffee Brewing</h1><h2>Beans</h2><h3>Roast</h3><h2>Methods</h2>
<h3>Pour Over</h3>'
heads = re.findall(r'<h([1-6])[^>]*>(.*?)</h\1>', html, re.S)
```

```
    for lvl, text in heads:
        print('  ' * (int(lvl) - 1) + text.strip())
```

```
output
Coffee Brewing
  Beans
    Roast
  Methods
    Pour Over
```

How it works:

1. The HTML contains a coffee-themed mix of H1, H2, and H3 headings in a sensible order.
2. re.findall captures the level digit and the inner text, using a backreference so the closing tag matches the opening level.
3. The loop multiplies two spaces by the level minus one to produce indentation, then prints the stripped text.
4. The result reads top to bottom as a nested outline, with deeper headings pushed further right.

Try it:

1. Add the heading level number as a prefix before each indented line
2. Color or mark any heading whose parent level was skipped over
3. Build the same outline from a real page fetched off disk
4. Indent with a '-' bullet prefix instead of spaces for a list-style outline.

⚠ Common Mistakes

- Without the `\1` back-reference, an `<h2>...</h3>` mismatch would still capture and misalign.
- Indenting by `level` instead of `level - 1` pushes the top-level h1 in by one stop.
- Nested inline tags inside a heading would leak into the captured text without extra stripping.

Skills gained backreference regex | indentation | outline rendering

→ Where You'll Use This: Auditors eyeball a rendered outline to spot lopsided nesting, orphaned subsections, or topics buried too deep before recommending a clearer heading structure.

Mental Model

It is the wireframe of a document's skeleton: indentation shows which bones hang off which, so a misshapen spine jumps out.

Project 156 Count Body Words After Stripping Tags

The Idea Behind It

Thin pages with little real text struggle to rank, so counting visible words starts by removing markup. This project replaces every tag with a space using a regex, then extracts runs of letters and apostrophes with findall, and reports the length of that word list as the body word count a reader would actually see.

What You'll Build & Learn

Strip all HTML tags from a page and count the words of visible body text that remain.

- Why thin content underperforms in search
- Stripping tags by replacing them with spaces
- Tokenising words with a letter-and-apostrophe pattern

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 155: build a heading outline. Project 157 changes the surface problem while preserving the underlying method.

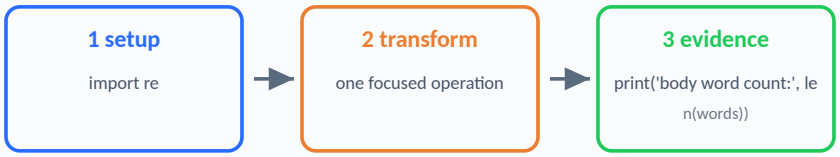
progress check

Reading target: count body words after stripping tags is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('body word count:', len(words))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`html = '<body><h1>Title</h1><p>The quick brown fox. It jumps over lazy dogs!</p><p>More words here, plenty more.</p></body>'`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print('body word count:', len(words))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
html = '<body><h1>Title</h1><p>The quick brown fox. It jumps over lazy dogs!</p>
<p>More words here, plenty more.</p></body>'
```

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Count Body Words After Stripping Tags: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, body, h1, Title. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: import re | Core: html = '<body><h1>Title</h1><p>The quick brown fox. It jumps over lazy... | Evidence: print('body word count:', len(words))

🛠️ Reader to Writer Bridge

1. Read: explain count body words after stripping tags in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 155.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 157 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('body word count:', len(words))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count body words after stripping tags and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`<[^>]+>` replaces every tag with a space so words never fuse across tag boundaries, then a word regex counts the remaining visible tokens.

```
import re
html = '<body><h1>Title</h1><p>The quick brown fox. It jumps over lazy dogs!
</p><p>More words here, plenty more.</p></body>'
text = re.sub(r'<[^>]+>', ' ', html)
words = re.findall(r"[A-Za-z']+", text)
print('body word count:', len(words))
```

```
output
body word count: 15
```

How it works:

1. The HTML body wraps a heading and two paragraphs of plain sentences.
2. re.sub replaces any run matching a tag pattern with a single space, collapsing markup into separators.

3. re.findall pulls out every token of letters and apostrophes, ignoring punctuation and the former tags.
4. len of that token list is printed as the body word count, which lands at fifteen here.

Try it:

1. Treat hyphenated words as one token rather than splitting them apart
2. Exclude text inside script and style tags before counting
3. Print the words-per-paragraph average alongside the total
4. Replace tags with '' and watch two real words merge into one.

⚠ Common Mistakes

- Replacing tags with '' instead of ' ' glues `</h1><p>` neighbours into one fake word.
- Counting `<script>` or `<style>` contents inflates the total with code, not prose.
- A word pattern of `[A-Za-z]+` alone would split contractions like `don't` into two words.

Skills gained tag stripping | word tokenising | content sizing

→ Where You'll Use This: Word-count thresholds drive thin-content reports; auditors use them to spot pages too light to compete for their target query and flag them for expansion.

⚡ Mental Model

Stripping tags is shaking the packaging off a parcel so you can weigh only the contents the customer actually receives.

Project 157 Confirm the Keyword Appears in Key Spots

💡 The Idea Behind It

A target term ideally appears in the title, the H1, and the opening paragraph, the three spots a reader and an engine weigh most. The code lowercases each of those captured regions using a fallback tuple when no match exists, then loops over the named pairs and prints yes or no for whether the keyword is present in each.

What You'll Build & Learn

Confirm a target keyword shows up in the title, the H1, and the first paragraph of a page.

- Keyword placement across primary elements
- A fallback tuple to avoid None on no match
- Membership testing inside lowercased text

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 156: count body words after stripping tags. Project 158 changes the surface problem while preserving the underlying method.

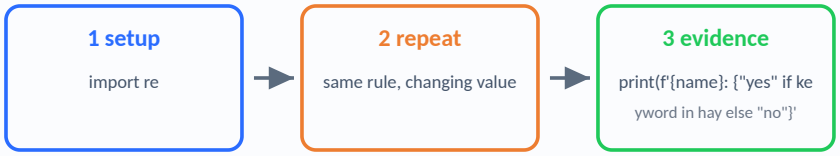
progress check

Reading target: confirm the keyword appears in key spots is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as data -> transformation -> evidence, not as isolated library calls.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{name}: {"yes" if keyword in hay else "no"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`print(f'{name}: {"yes" if keyword in hay else "no"}')`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print(f'{name}: {"yes" if keyword in hay else "no"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{name}: {"yes" if keyword in hay else "no"}')`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Confirm the Keyword Appears in Key Spots: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, title, Trail, Running. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: print(f'{name}: {"yes" if keyword in hay else "no"}') | Evidence: print(f'{name}: {"yes" if keyword in hay else "no"}')

Reader to Writer Bridge

1. Read: explain confirm the keyword appears in key spots in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 156.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 158 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{name}: {"yes" if keyword in hay else "no"}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of confirm the keyword appears in key spots and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each element is lowercased so the membership test is case-insensitive; the `or [None, '']` fallback keeps the code safe when a tag is missing instead of crashing on `.group`.

```
import re
html = '<title>Trail Running Shoes Review</title><h1>Best Trail Running
Shoes</h1><p>Our guide covers grip, cushioning and fit for every runner.</p>'
keyword = 'trail running shoes'
title = (re.search(r'<title>(.*?)</title>', html, re.S) or [None, '']
)[1].lower()
h1 = (re.search(r'<h1[^>]*>(.*?)</h1>', html, re.S) or [None, '']
)[1].lower()
first_p = (re.search(r'<p[^>]*>(.*?)</p>', html, re.S) or [None, '']
)[1].lower()
for name, hay in [('title', title), ('h1', h1), ('first paragraph', first_p)]:
    print(f'{name}: {"yes" if keyword in hay else "no"}')
```

```
output
title: yes
h1: yes
first paragraph: no
```

How it works:

1. The HTML supplies a title and H1 that mention trail running shoes plus a paragraph that does not.
2. Each region is captured by its own regex, and an or with a two-item fallback list yields an empty string when search returns None.
3. Indexing element one of either the match or the fallback gives the text, which is lowercased for case-insensitive matching.
4. A loop over the named pairs prints yes when the keyword substring is found, no otherwise, exposing the missing first paragraph.

Try it:

1. Add a check for the keyword inside any H2 on the page

2. Match individual keyword words rather than the exact phrase as one
3. Report a single placement score out of the three regions checked
4. Reword the first paragraph to include the keyword and watch its flag flip to 'yes'.

⚠ Common Mistakes

- Comparing without lowercasing makes 'Trail' and 'trail' miss each other.
- A substring test matches the keyword even when it is split across other words, which can over-report.
- Indexing a failed `re.search` with `[1]` only works because of the fallback list; calling `.group(1)` on `None` would crash.

Skills gained substring search | fallback handling | placement check

→ Where You'll Use This: On-page optimisation guides recommend the keyword in title, H1, and intro; this triad check tells a writer at a glance which prime location still needs the target phrase.

⚡ Mental Model

It is a pre-flight checklist of three gauges; you want all three needles in the green before the page takes off.

Project 158 Find Images Missing an Alt Attribute

💡 The Idea Behind It

An image with no alt attribute is silent to screen readers and gives crawlers no text to index. This project finds every img tag with a word-boundary regex, keeps only those whose tag has no alt assignment, and reports how many images exist, how many lack alt, and prints each offending tag.

What You'll Build & Learn

Scan all image tags on a page and list the ones missing an alt attribute entirely.

- Missing alt hurts accessibility and image SEO
- Matching whole img tags with a word boundary
- Filtering a list by absence of a pattern

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 157: confirm the keyword appears in key spots. Project 159 changes the surface problem while preserving the underlying method. *progress check*

Reading target: find images missing an alt attribute is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(' ', t)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`missing = [t for t in imgs if not re.search(r'\balt\s*=', t)]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' ', t)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [t for t in imgs if not re.search(r'\balt\s*=', t)]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Images Missing an Alt Attribute: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, img, src, hero. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: missing = [t for t in imgs if not re.search(r'\balt\s*=', t)] | Evidence: print(' ', t)

🛠️ Reader to Writer Bridge

1. Read: explain find images missing an alt attribute in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 157.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 159 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' ', t)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of find images missing an alt attribute and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`<img\b[^>]*>` captures each image tag, and the `\balt\s*=` test distinguishes a tag with no alt attribute at all from one that merely has an empty alt.

```
import re
html = '<img src="hero.jpg" alt="Runner on a trail"><img src="logo.png"><img
src="chart.svg" alt="">'
imgs = re.findall(r'<img\b[^>]*>', html)
missing = [t for t in imgs if not re.search(r'\balt\s*=', t)]
print('img tags:', len(imgs))
print('missing alt:', len(missing))
for t in missing:
    print(' ', t)
```

```
output
img tags: 3
missing alt: 1
  <img src="logo.png">
```

How it works:

1. The HTML has three images: one with descriptive alt, one with no alt, and one with an empty alt.
2. re.findall with the img word-boundary pattern returns each full image tag as a string.
3. A comprehension keeps tags where a search for alt equals finds nothing, isolating the truly missing ones.
4. The script prints the total tag count, the missing count, and each offending tag indented underneath.

Try it:

1. Pull the src of each offending image so it is easy to locate
2. Count images that have an alt attribute as a separate total
3. Treat a self-closing slash before the bracket the same way
4. Add `alt="Logo"` to the second image and watch the missing count drop to zero.

⚠ Common Mistakes

- An empty `alt=""` is present (and intentional for decorative images), so it must NOT count as missing.
- Without the `\b` word boundary, an attribute like `data-alt` would falsely satisfy the alt check.
- Self-closing `<img .../>` still matches `[^>]*>` here, but a malformed unclosed tag would not.

Skills gained `img matching` `alt audit` `list filtering`

→ **Where You'll Use This:** Missing-alt reports are a staple of accessibility scans and image-SEO checks; fixing them lets engines understand pictures and serves users on screen readers.

⚡ Mental Model

An image without alt is a photo in an album with no caption: the sighted skim past it, but anyone listening hears only silence.

Project 159 Tell Empty Alt From Decorative Intent

💡 The Idea Behind It

Not every image needs description: a spacer or decoration should carry an empty alt so screen readers skip it, while a missing alt is a real fault. The code inspects each img tag, reads its alt and src, and sorts it into described, decorative empty alt, or missing, distinguishing intentional emptiness from oversight.

What You'll Build & Learn

Classify each image as described, intentionally decorative, or missing its alt attribute outright.

- Empty alt is valid for decorative images
- Three-way classification beyond present or absent
- Reading two attributes from one tag

🧭 Skill Roadmap

`Skill step:` adds one small variation while the chapter pattern is still new. It builds on Project 158: find images missing an alt attribute. Project 160 changes the surface problem while preserving the underlying method.

progress check

`Reading target:` tell empty alt from decorative intent is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- `Writing move:` this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print(f'{src}: {kind}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if m is None:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{src}: {kind}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m is None:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Tell Empty Alt From Decorative Intent: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, img, src, jpg. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if m is None: | Evidence: print(f'{src}: {kind}')

🛠️ Reader to Writer Bridge

1. Read: explain tell empty alt from decorative intent in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 158.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 160 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{src}: {kind}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of tell empty alt from decorative intent and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A captured alt value that is `None` means the attribute is absent, while a captured value that strips to nothing is a deliberate decorative marker; only non-blank text counts as described.

```
import re
html = '<img src="a.jpg" alt="Product photo"><img src="spacer.gif" alt=""><img
src="b.png" alt="   ">'
for tag in re.findall(r'<img\b[^>]*>', html):
    m = re.search(r'\balt\s*=\s*"([^"]*)"', tag)
    src = re.search(r'src\s*=\s*"([^"]*)"', tag).group(1)
    if m is None:
        kind = 'MISSING alt'
    elif m.group(1).strip() == '':
        kind = 'empty alt (decorative)'
    else:
        kind = 'described'
    print(f'{src}: {kind}')
```

```
output
a.jpg: described
spacer.gif: empty alt (decorative)
b.png: empty alt (decorative)
```

How it works:

1. The HTML holds a described product photo, a spacer with empty alt, and an image whose alt is only spaces.
2. For each img tag the code searches for an alt value and separately captures the src.
3. When the alt search is None the kind is missing, when its stripped value is empty the kind is decorative, otherwise described.
4. Each image's src and classification are printed, with the whitespace-only alt correctly read as decorative.

Try it:

1. Add a fourth category for alt text that is suspiciously long
2. Flag a decorative image whose filename suggests it is actually meaningful
3. Count how many images fall into each of the three buckets
4. Give `spacer.gif` real alt text and watch it move to 'described'.

⚠ Common Mistakes

- Whitespace-only alt (`alt=" "`) is effectively empty; not stripping it would mislabel it 'described'.
- Distinguishing `None` (no attribute) from `''` (empty attribute) is the whole point; collapsing them loses the decorative signal.
- An alt with only a stray entity like ` ` would be treated as text here even though it renders blank.

Skills gained attribute parsing alt classification intent detection

→ Where You'll Use This: Mature accessibility audits separate genuine omissions from valid decorative markup so teams fix the real problems without flooding reports with intentional empty alts.

Mental Model

It is triage at a clinic: the described patients are fine, the decorative ones are deliberately resting, and only the missing ones need urgent care.

Project 160 Verify a Canonical Link Is Present

The Idea Behind It

A canonical link tells engines which URL is the master copy, heading off duplicate-content dilution. The code searches for a link tag whose rel is canonical, and when found it digs the href value out of that exact tag and prints the canonical URL; otherwise it reports the canonical as missing.

What You'll Build & Learn

Confirm a page declares a canonical link and read the URL it points to.

- What a canonical tag prevents
- Matching one specific link among several
- Two-stage extraction of tag then attribute

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 159: tell empty alt from decorative intent. Project 161 changes the surface problem while preserving the underlying method.

progress check

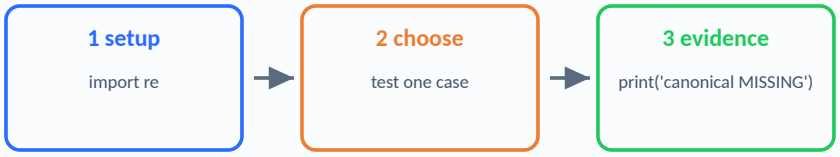
Reading target: verify a canonical link is present is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print('canonical MISSING')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if m:`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('canonical MISSING')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m:`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Verify a Canonical Link Is Present: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, head, link, rel. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if m: | Evidence: print('canonical MISSING')

Reader to Writer Bridge

1. Read: explain verify a canonical link is present in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 159.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 161 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print('canonical MISSING')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of verify a canonical link is present and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

The outer regex isolates the single `<link>` tag whose rel is canonical (ignoring the stylesheet link), then a second pass pulls the href out of just that tag.

```
import re
html = '<head><link rel="stylesheet" href="/s.css"><link rel="canonical"
href="https://example.com/blog/post"></head>'
m = re.search(r'<link\b[^>]*rel\s*=\s*"canonical"[^>]*>', html)
if m:
    href = re.search(r'href\s*=\s*"([^"]*)"', m.group(0)).group(1)
    print('canonical present:', href)
```

```
else:
    print('canonical MISSING')
```

```
output
canonical present: https://example.com/blog/post
```

How it works:

1. The head holds a stylesheet link and a canonical link pointing at a blog post URL.
2. The first regex matches a link tag that contains rel equals canonical, returning the whole tag.
3. A second regex run over that matched tag captures the href value between its quotes.
4. The script prints the canonical URL when present, or a missing message when the first search fails.

Try it:

1. Warn when the canonical URL points to a different domain than expected
2. Detect more than one canonical tag, which is itself an error
3. Compare the canonical against the page's own address for self-reference
4. Remove the canonical link and confirm the 'MISSING' branch runs.

⚠ Common Mistakes

- Searching the whole document for any href would grab the stylesheet's href instead of the canonical's.
- A canonical pointing at a relative path still matches, but search engines prefer an absolute URL.
- Matching `rel="canonical"` exactly misses a tag that lists multiple rel values like `rel="canonical alternate"`.

Skills gained canonical check | nested extraction | link tag regex

→ Where You'll Use This: Duplicate-content audits verify every page declares a sensible canonical; a missing or wrong canonical is a frequent cause of the wrong URL being indexed or ranking.

Mental Model

The canonical is a redirect signpost among identical doors saying this one is the real entrance, so visitors do not split across copies.

Project 161 Parse the Meta Robots Directives

The Idea Behind It

A meta robots tag governs whether a page may be indexed and its links followed. This project captures the robots content value, splits it on commas into lowercased directives, then derives indexable and followable booleans by checking for noindex and nofollow, surfacing whether the page is quietly being suppressed.

What You'll Build & Learn

Parse a meta robots tag into directives and report whether the page is indexable and followable.

- How meta robots controls indexing
- Splitting a comma list into clean tokens
- Deriving booleans from membership tests

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 160: verify a canonical link is present. Project 162 changes the surface problem while preserving the underlying method.

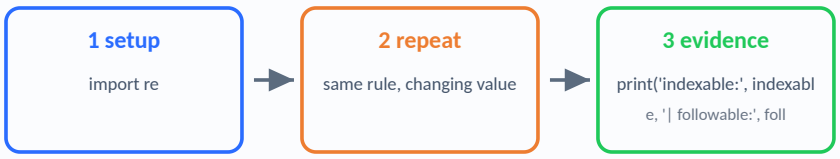
progress check

Reading target: parse the meta robots directives is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('indexable:', indexable, '| followable:', followable)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit

Move	Code to inspect	Why it matters
		titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`directives = [d.strip().lower() for d in m.group(1).split(',')] if m else []`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('indexable:', indexable, '| followable:', followable)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `directives = [d.strip().lower() for d in m.group(1).split(',')] if m else []`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Parse the Meta Robots Directives: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, meta, name, robots. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: directives = [d.strip().lower() for d in m.group(1).split(',')] if m el... | Evidence: print('indexable:', indexable, '| followable:', followable)

Reader to Writer Bridge

1. Read: explain parse the meta robots directives in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 160.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 162 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('indexable:', indexable, '| followable:', followable)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of parse the meta robots directives and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The content value is split on commas and normalised, then indexability is decided by the absence of `noindex` and `nofollow` rather than the presence of their positive opposites.

```
import re
html = '<meta name="robots" content="index, follow, max-snippet:-1">'
m = re.search(r'<meta\s+name="robots"\s+content="([^"]*)"', html)
directives = [d.strip().lower() for d in m.group(1).split(',')] if m else []
indexable = 'noindex' not in directives
followable = 'nofollow' not in directives
print('directives:', directives)
print('indexable:', indexable, '| followable:', followable)
```

```
output
directives: ['index', 'follow', 'max-snippet:-1']
indexable: True | followable: True
```

How it works:

1. The HTML has a robots meta whose content lists index, follow, and a max-snippet directive.
2. The regex captures the content value, which is split on commas with each piece stripped and lowercased into a directives list.
3. indexable is true when noindex is absent and followable is true when nofollow is absent.
4. The script prints the directive list and the two derived booleans.

Try it:

1. Also surface the max-snippet value as a separate number
2. Default to indexable and followable when the robots tag is absent
3. Recognise the all and none shorthand directives explicitly
4. Change the content to `noindex, nofollow` and watch both flags go False.

⚠ Common Mistakes

- A missing robots tag means default index/follow, so an empty directive list must read as indexable.
- Checking for `index` to decide indexability is wrong; the real rule is the absence of `noindex`.
- Forgetting to strip and lowercase makes ` Noindex` slip past the membership test.

Skills gained directive parsing | string split | boolean logic

→ Where You'll Use This: A stray noindex left in a template can silently de-index whole sections; auditors parse robots directives to catch pages accidentally hidden from search.

⚡ Mental Model

The robots tag is a doorman's note: index and follow waves engines in, while noindex quietly tells them to walk on by.

Project 162 Detect Open Graph and Twitter Card Tags

💡 The Idea Behind It

Open Graph and Twitter Card meta tags decide how a page looks when shared on social platforms. The code collects the set of og properties and the set of twitter card names with two findall calls, prints each sorted set, then checks specifically whether the commonly forgotten og:description property is present.

What You'll Build & Learn

List a page's Open Graph and Twitter Card meta tags and point out a commonly missing one.

- Social meta tags shape share previews
- Deduplicating matches with a set
- Checking for one expected key by membership

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 161: parse the meta robots directives. Project 163 changes the surface problem while preserving the underlying method.

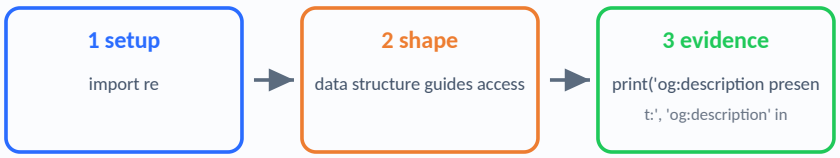
progress check

Reading target: detect open graph and twitter card tags is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The data structure is doing part of the thinking. Ask what the collection stores, how items are found, and what order or uniqueness it promises.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('og:description present:', 'og:description' in og)` exposes the result; match it against the output before changing the program. output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, shape applies the chapter idea, and evidence proves your trace against the output. visual trace

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`'<meta property="og:image" content="https://example.com/a.jpg">'`	This line depends on the chosen data shape. Ask how items are stored, found, or ordered. slow line
3. Check the evidence	`print('og:description present:', 'og:description' in og)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `'<meta property="og:image" content="https://example.com/a.jpg">'`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Name the collection shape, then ask how an item is added, selected, transformed, or counted.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Detect Open Graph and Twitter Card Tags: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, meta, property, og. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: '<meta property="og:image" content="https://example.com/a.jpg">' | Evidence: print('og:description present:', 'og:description' in og)

🛠️ Reader to Writer Bridge

1. Read: explain detect open graph and twitter card tags in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 161.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 163 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('og:description present:', 'og:description' in og)`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect open graph and twitter card tags and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Open Graph tags live in the `property` attribute while Twitter tags use `name`; collecting each into a set lets the audit check coverage and spot a missing recommended tag.

```
import re
html = ('<meta property="og:title" content="Trail Shoes">'
        '<meta property="og:image" content="https://example.com/a.jpg">'
        '<meta name="twitter:card" content="summary_large_image">')
og = set(re.findall(r'property="(og:[^"]+)"', html))
tw = set(re.findall(r'name="(twitter:[^"]+)"', html))
print('open graph:', sorted(og))
print('twitter card:', sorted(tw))
print('og:description present:', 'og:description' in og)
```

```
output
open graph: ['og:image', 'og:title']
twitter card: ['twitter:card']
og:description present: False
```

How it works:

1. A multi-line string concatenates og:title, og:image, and a twitter:card meta tag.
2. One findall captures every og property name into a set, another captures twitter names into a second set.
3. Both sets are printed sorted so the present tags read in order.
4. A membership test reports og:description as absent, the typical gap in social markup.

Try it:

1. Report which required og tags are missing rather than just one
2. Also capture the content value behind each og:image tag
3. Add detection for the twitter:image and twitter:title tags
4. Add an `og:description` meta and watch the final check flip to True.

⚠ Common Mistakes

- Searching `name=` for Open Graph misses them because OG uses the `property` attribute.
- Printing the raw sets without sorting makes the output order vary between runs.
- A page can have OG tags but no Twitter card, which still shares acceptably via OG fallbacks.

Skills gained set building | social tags | presence check

→ Where You'll Use This: Social-sharing audits verify Open Graph and card tags so links render with a title, image, and description instead of a bare URL across Facebook, LinkedIn, and X.

⚡ Mental Model

Social tags are the trading-card art for your link: leave a slot blank and the card shows up dull where rivals look vivid.

Project 163 Flag Empty or Missing Titles Across Pages

💡 The Idea Behind It

A title that is absent or blank squanders the strongest on-page lever, and the two faults look different in markup. This project loops over a small set of pages, searches each for a title, and distinguishes no title tag at all from a present-but-empty title, printing a clear verdict per path.

What You'll Build & Learn

Audit several pages and separate the ones missing a title tag from the ones whose title is empty.

- Absent versus empty are distinct title faults
- Iterating a dictionary of pages by sorted key
- Branching on None versus a blank capture

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 162: detect open graph and twitter card tags. Project 164 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag empty or missing titles across pages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{path}: ok ({m.group(1)!r})')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if m is None:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: ok ({m.group(1)!r})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m is None:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Empty or Missing Titles Across Pages: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, pages, head, title, Welcome. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if m is None: | Evidence: print(f'{path}: ok ({m.group(1)!r})')

🛠️ Reader to Writer Bridge

1. Read: explain flag empty or missing titles across pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 162.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 164 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: ok ({m.group(1)!r})')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of flag empty or missing titles across pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A failed match (`None`) means the tag is absent, while a match whose stripped text is empty means the tag exists but is blank; the two faults need different fixes so the audit keeps them distinct.

```
import re
pages = {
    '/a': '<head><title>Welcome</title></head>',
    '/b': '<head><title></title></head>',
    '/c': '<head><meta charset="utf-8"></head>',
}
for path in sorted(pages):
    m = re.search(r'<title>(.*?)</title>', pages[path], re.S)
    if m is None:
        print(f'{path}: NO title tag')
    elif m.group(1).strip() == '':
        print(f'{path}: EMPTY title')
    else:
        print(f'{path}: ok ({m.group(1)!r})')
```

```
output
/a: ok ('Welcome')
/b: EMPTY title
/c: NO title tag
```

How it works:

1. A pages dictionary maps three paths to markup: a good title, an empty title, and a head with no title.
2. Iterating the sorted keys, each page is searched for a title with re.S so newlines do not break it.
3. When search returns None the path has no title tag; when the stripped capture is empty the title is blank.
4. Otherwise the path prints ok with the title shown via the repr conversion.

Try it:

1. Add a fourth case flagging a title that is only whitespace
2. Collect the offending paths into a list and print a summary count
3. Read the page markup from files in a directory instead of a literal dict
4. Add a page whose title is just spaces and confirm it reads EMPTY.

⚠ Common Mistakes

- Collapsing 'no tag' and 'empty tag' into one bucket hides that the fixes differ.

- Iterating the dict without sorting the keys makes the report order nondeterministic.
- A title of only whitespace passes a truthiness check but is still effectively empty.

Skills gained multi-page audit | None handling | empty detection

→ Where You'll Use This: Site-wide title checks are core to any crawl report; both blank and absent titles leave engines guessing a page's topic, so audits surface each separately for fixing.

⚡ Mental Model

Scanning titles is checking name tags at a conference: some are missing entirely, some are pinned on blank, and both leave you unintroduced.

Project 164 Compute Keyword Density

💡 The Idea Behind It

Keyword density is the share of body words equal to the target term, where a modest level signals focus and a sky-high one reads as stuffing. The code strips tags, lowercases the text, tokenises letter runs, counts occurrences of the keyword, and computes density as a rounded percentage of the total word count.

What You'll Build & Learn

Compute the percentage of body words that equal a target keyword to gauge topical focus.

- Density as keyword hits over total words
- Lowercasing before counting for case fairness
- Rounding a percentage with round

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 163: flag empty or missing titles across pages. Project 165 changes the surface problem while preserving the underlying method.

progress check

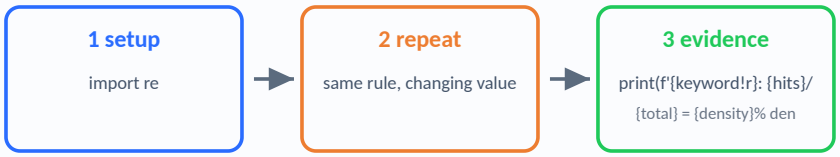
Reading target: compute keyword density is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{keyword!r}: {hits}/{total} = {density}% density')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`html = '<h1>Running Shoes</h1><p>These running shoes suit road and trail runners. Shoes built for speed.</p>'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{keyword!r}: {hits}/{total} = {density}% density')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
html = '<h1>Running Shoes</h1><p>These running shoes suit road and trail runners. Shoes built for speed.</p>'
```

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute Keyword Density: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, collections, Counter, html, h1. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: html = '<h1>Running Shoes</h1><p>These running shoes suit road and trai... | Evidence: print(f'{keyword!r}: {hits}/{total} = {density}% density')

🛠️ Reader to Writer Bridge

1. Read: explain compute keyword density in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 163.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 165 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{keyword!r}: {hits}/{total} = {density}% density')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute keyword density and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

After stripping tags and lowercasing, density is simply the keyword's count divided by total word count; rounding keeps the printed float stable and deterministic.

```
import re
from collections import Counter
html = '<h1>Running Shoes</h1><p>These running shoes suit road and trail
runners. Shoes built for speed.</p>'
text = re.sub(r'<[^>]+>', ' ', html).lower()
words = re.findall(r"[a-z]+", text)
total = len(words)
keyword = 'shoes'
hits = words.count(keyword)
density = round(100 * hits / total, 1)
print(f'{keyword!r}: {hits}/{total} = {density}% density')
```

```
output
'shoes': 3/14 = 21.4% density
```

How it works:

1. The HTML mixes a heading and paragraph where the word shoes appears three times.
2. re.sub removes tags and the text is lowercased, then findall pulls every lowercase letter run as a word.
3. total is the word count, hits is the list count of the keyword, and density is one hundred times hits over total, rounded.
4. The script prints the keyword with its hit ratio and the density percentage.

Try it:

1. Count a two-word phrase rather than a single keyword token
2. Flag the density as stuffed once it climbs above a chosen threshold
3. Report the densities of the three most frequent content words
4. Switch the keyword to a two-word phrase and count phrase occurrences instead.

⚠ Common Mistakes

- Including tag names or attribute text in the word list distorts the denominator.
- Counting `shoes` would also catch `shoeshine` with a substring test; `==` on tokens avoids that.

- Dividing without rounding prints a long, noisy float that varies in appearance.

Skills gained density calc | word counting | Counter-free tally

→ Where You'll Use This: Content tools report keyword density so writers avoid both starving a page of its target term and overusing it to the point that engines see manipulation.

Mental Model

Density is seasoning in a dish: a pinch of the keyword brings out the flavour, but dumping the whole jar ruins the meal.

Project 165 Measure the Stop-Word Ratio

The Idea Behind It

When most of a page's words are filler like the and of, there is little substance for a crawler to digest. This project defines a stop-word set, strips tags and tokenises the lowercased text, keeps only the words found in that set, and reports their share of the total as a stop-word ratio that proxies content richness.

What You'll Build & Learn

Calculate the fraction of a page's words that are stop words as a quick richness proxy.

- Stop words carry little topical meaning
- Membership filtering against a set
- Ratio of a subset to the whole

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 164: compute keyword density. Project 166 changes the surface problem while preserving the underlying method. *progress check*

Reading target: measure the stop-word ratio is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print(f'stop words: {len(stops)}/{len(words)} = {ratio}%')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`stops = [w for w in words if w in stop]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'stop words: {len(stops)}/{len(words)} = {ratio}%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `stops = [w for w in words if w in stop]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure the Stop-Word Ratio: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, The, best, of. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: stops = [w for w in words if w in stop] | Evidence: print(f'stop words: {len(stops)}/{len(words)} = {ratio}%')

🛠️ Reader to Writer Bridge

1. Read: explain measure the stop-word ratio in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 164.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 166 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'stop words: {len(stops)}/{len(words)} = {ratio}%')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure the stop-word ratio and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering the lowercased word list against a stop-word set and dividing by the total gives the proportion of low-information words on the page.

```
import re
html = '<p>The best of the running shoes are on the shelf and in the store
today.</p>'
stop = {'the', 'of', 'are', 'on', 'and', 'in', 'a', 'to', 'is'}
words = re.findall(r"[a-z]+", re.sub(r'<[^>]+>', ' ', html).lower())
stops = [w for w in words if w in stop]
ratio = round(100 * len(stops) / len(words), 1)
print(f'stop words: {len(stops)}/{len(words)} = {ratio}%')
```

```
output
stop words: 9/15 = 60.0%
```

How it works:

1. The HTML paragraph is deliberately padded with articles and prepositions.
2. A stop set lists common function words, and the text is stripped of tags, lowercased, and tokenised into words.
3. A comprehension keeps only words present in the stop set, building the stops list.
4. The ratio is one hundred times the stop count over the total word count, rounded, and printed with the raw counts.

Try it:

1. Expand the stop set with more common English function words
2. Print the most frequent non-stop word as the apparent topic
3. Invert the metric to report the content-word ratio instead
4. Add 'for' and 'with' to the stop set and watch the ratio rise.

⚠ Common Mistakes

- A short or incomplete stop list undercounts and makes a stuffed page look content-rich.
- Case sensitivity makes a capitalised 'The' at a sentence start slip past the set.
- Dividing by zero crashes on an empty body, so an empty page needs a guard.

Skills gained stop words | set membership | ratio calc

→ Where You'll Use This: A high stop-word ratio hints at fluffy or padded copy; content audits use it as a fast signal to mark pages that say very little of substance.

Mental Model

Stop words are the foam on a poured beer: a little is normal, but a glass that is mostly froth leaves you barely any drink.

Project 166 Estimate Readability by Sentence Length

The Idea Behind It

Long sentences read harder, and readability is itself an on-page quality signal. The code strips tags, splits the text on sentence-ending punctuation while discarding empty fragments, tokenises the words, and divides word count by sentence count to give an average words-per-sentence figure with no external dependency.

What You'll Build & Learn

Estimate prose readability by averaging the number of words per sentence on a page.

- Average sentence length as a readability proxy
- Splitting on sentence-ending punctuation
- Discarding empty fragments after a split

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 165: measure the stop-word ratio. Project 167 changes the surface problem while preserving the underlying method.

progress check

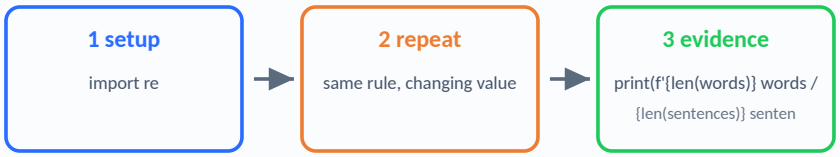
Reading target: estimate readability by sentence length is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{len(words)} words / {len(sentences)} sentences = {avg} words/sentence')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`sentences = [s for s in re.split(r'[.!?]+', text) if s.strip()]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{len(words)} words / {len(sentences)} sentences = {avg} words/sentence')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sentences = [s for s in re.split(r'[.!?]+', text) if s.strip()]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Estimate Readability by Sentence Length: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, Running, is, great. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: sentences = [s for s in re.split(r'[.!?]+', text) if s.strip()] | Evidence: print(f'{len(words)} words / {len(sentences)} sentences = {avg} words/s...

🛠️ Reader to Writer Bridge

1. Read: explain estimate readability by sentence length in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 165.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 167 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{len(words)} words / {len(sentences)} sentences = {avg} words/sentence')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of estimate readability by sentence length and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Splitting on sentence-ending punctuation and discarding empty fragments counts sentences, and dividing the word total by that count yields average sentence length.

```
import re
html = '<p>Running is great. It builds endurance and strength over time! Are
you ready to start today?</p>'
text = re.sub(r'<[^>]+>', ' ', html)
```

```
sentences = [s for s in re.split(r'[.!?]+', text) if s.strip()]
words = re.findall(r"[A-Za-z']+", text)
avg = round(len(words) / len(sentences), 1)
print(f'{len(words)} words / {len(sentences)} sentences = {avg}
words/sentence')
```

```
output
16 words / 3 sentences = 5.3 words/sentence
```

How it works:

1. The HTML paragraph holds three short sentences ended by a period, exclamation, and question mark.
2. re.sub strips tags, then re.split on one or more of period, bang, or question breaks the text into sentence fragments.
3. A comprehension drops fragments that are only whitespace, and findall counts the words across the text.
4. Dividing words by sentences and rounding gives the average, printed as words per sentence.

Try it:

1. Flag the page as hard to read above a chosen average length
2. Count syllables to approximate a fuller readability grade
3. Report the single longest sentence found on the page
4. Add a much longer sentence and watch the average climb.

⚠ Common Mistakes

- Splitting on '.' alone treats 'Inc.' or '3.5' as a sentence break and inflates the count.
- An empty trailing fragment after the final '!' would count as a sentence without the strip filter.
- Abbreviations and ellipses can fool the naive punctuation split in real text.

Skills gained sentence split | readability | averaging

→ Where You'll Use This: Readability scores guide editors toward shorter, scannable sentences; this dependency-free average stands in for heavier formulas when a quick signal is enough.

Mental Model

Sentence length is the gear you climb a hill in: short ones let readers keep pace, while one endless clause stalls them on the slope.

Project 167 Count Internal Versus External Links

The Idea Behind It

The mix of internal links within your site versus external ones shapes how authority flows. This project walks every href, counts a leading slash as internal, and for absolute URLs extracts the domain and treats it as internal when it equals the host or is a subdomain of it, otherwise external, revealing whether a page hoards or leaks equity.

What You'll Build & Learn

Classify each link on a page as internal or external and tally the two counts.

- Link equity flow between internal and external
- Recognising relative versus absolute URLs
- Subdomain matching with endswith

🧭 Skill Roadmap

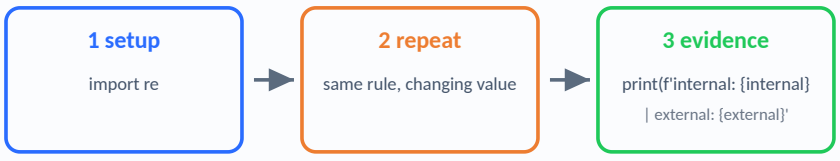
Skill step: moves you from recognition toward independent tracing. It builds on Project 166: estimate readability by sentence length. Project 168 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count internal versus external links is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'internal: {internal} | external: {external}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if href.startswith('/'):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'internal: {internal} | external: {external}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if href.startswith('/'):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Internal Versus External Links: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, host, example, com, html. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if href.startswith('/'): | Evidence: print(f'internal: {internal} | external: {external}')

🛠️ Reader to Writer Bridge

1. Read: explain count internal versus external links in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 166.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 168 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'internal: {internal} | external: {external}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count internal versus external links and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A leading-slash href is internal by definition, and an absolute URL is internal when its host equals the site domain or is a subdomain of it (the `.` + host suffix test).

```
import re
host = 'example.com'
html = ('<a href="https://example.com/about">About</a>'
        '<a href="/blog">Blog</a>'
        '<a href="https://other.com/ref">Partner</a>'
        '<a href="https://shop.example.com/cart">Cart</a>')
internal = external = 0
for href in re.findall(r'href\s*=\s*"([^"]*)"', html):
    if href.startswith('/'):
        internal += 1
    else:
        dom = href.split('://', 1)[1].split('/', 1)[0]
        if dom == host or dom.endswith('.' + host):
            internal += 1
        else:
            external += 1
print(f'internal: {internal} | external: {external}')
```

```
output
internal: 3 | external: 1
```

How it works:

1. The HTML lists four anchors: an absolute same-domain link, a relative link, an external link, and a subdomain link.
2. Each href is examined; one starting with a slash is internal immediately.
3. For absolute URLs the domain is sliced out after the scheme and before the next slash, then compared to the host.
4. A domain equal to the host or ending in a dot plus the host counts as internal, else external, and both totals are printed.

Try it:

1. Print the actual external domains that the page links out to
2. Handle protocol-relative URLs that begin with two slashes
3. Separately count links that point back to the page itself
4. Add a `mailto:` link and decide which bucket (or neither) it belongs in.

⚠ Common Mistakes

- Treating a subdomain like `shop.example.com` as external miscounts your own pages.
- A naive `host in href` test wrongly counts `notexample.com` as internal.
- Protocol-relative or mailto links would need their own branch before the split.

Skills gained link classification | URL parsing | host matching

→ Where You'll Use This: Internal-linking audits use this split to ensure pages distribute authority through the site rather than bleeding it all to outside domains or hoarding it.

⚡ Mental Model

Internal links are pipes routing water around your own house; external ones drain it to the neighbours, so you watch the balance.

Project 168 Flag Links With Empty Anchor Text

💡 The Idea Behind It

Anchor text tells engines what a linked page is about, so a link with no readable text wastes that signal, though an image link can rescue it through alt text. The code captures each anchor's href and inner markup, strips tags from the inner to get text, falls back to an image alt, and flags the link when neither yields a label.

What You'll Build & Learn

Find links whose anchor text is empty, crediting an image link whose alt supplies a label.

- Anchor text as a relevance signal
- Falling back to image alt for a label
- Stripping inner tags to recover text

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 167: count internal versus external links. Project 169 changes the surface problem while preserving the underlying method.

progress check

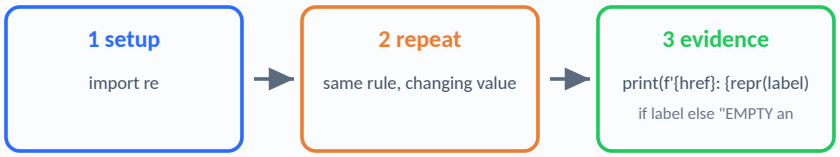
Reading target: flag links with empty anchor text is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{href}: {repr(label) if label else "EMPTY anchor text"}')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`label = text or (alt.group(1).strip() if alt else '')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{href}: {repr(label) if label else "EMPTY anchor text"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `label = text or (alt.group(1).strip() if alt else '')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Links With Empty Anchor Text: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, href, shoes, Trail. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: label = text or (alt.group(1).strip() if alt else '') | Evidence: print(f'{href}: {repr(label) if label else "EMPTY anchor text"}')

🛠️ Reader to Writer Bridge

1. Read: explain flag links with empty anchor text in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 167.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 169 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{href}: {repr(label) if label else "EMPTY anchor text"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag links with empty anchor text and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Inner tags are stripped to find visible text, and when that is blank the alt text of a contained image is used as a fallback label before declaring the anchor empty.

```
import re
html = ('<a href="/shoes">Trail shoes</a>'
        '<a href="/sale"></a>'
        '<a href="/img"><img src="x.jpg" alt="Boot guide"></a>'
        '<a href="/blank">   </a>')
for m in re.finditer(r'<a\b[^>]*href="([^"]*)"[^>]*>(.*?)</a>', html, re.S):
    href, inner = m.group(1), m.group(2)
    text = re.sub(r'<[^>]+>', '', inner).strip()
    alt = re.search(r'<img\b[^>]*\balt="([^"]*)"', inner)
    label = text or (alt.group(1).strip() if alt else '')
    print(f'{href}: {repr(label) if label else "EMPTY anchor text"}')
```

```
output
/shoes: 'Trail shoes'
/sale: EMPTY anchor text
/img: 'Boot guide'
/blank: EMPTY anchor text
```

How it works:

1. The HTML has four anchors: a text link, an empty one, an image link with alt, and a whitespace-only link.
2. finditer captures each anchor's href and its inner content between the tags.
3. The inner markup is stripped of tags and trimmed to text, and an image alt is searched as a fallback label.
4. label takes the text or the alt, and the script prints the href with its label or an empty-anchor warning.

Try it:

1. Warn on generic anchor text like click here or read more
2. Collect every empty-anchor href into a list for a summary
3. Strip extra inline tags like span inside the anchor text
4. Remove the image's alt and watch `/img` become an empty anchor.

⚠ Common Mistakes

- Treating an image-only link as empty ignores that its alt text is the effective anchor.
- A non-greedy `(.*?)` is essential so the regex stops at the first `</a>`, not the last.
- Whitespace-only inner text must be stripped or `/blank` would look labelled.

Skills gained anchor audit | finditer | alt fallback

→ Where You'll Use This: Link audits flag bald links so editors add descriptive anchor text; correctly crediting image-link alt keeps the report from false-flagging legitimate icon and logo links.

⚡ Mental Model

Anchor text is the label on a hallway door; a blank one leaves engines guessing the room, while the image's alt is a handwritten sticky note.

Project 169 Count Nofollow Versus Followed Links

💡 The Idea Behind It

A nofollow link asks engines not to pass ranking credit, so counting followed against nofollow shows how much equity actually flows onward. This project reads each anchor's rel attribute, splits it into tokens, and increments nofollow when that token appears, otherwise treating the link as a followed, equity-passing link.

What You'll Build & Learn

Count how many links are nofollow versus followed to see how much link equity flows on.

- Nofollow withholds ranking credit
- Splitting a rel value into multiple tokens

- Counting two outcomes in one loop

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 168: flag links with empty anchor text. Project 170 changes the surface problem while preserving the underlying method.

progress check

Reading target: count nofollow versus followed links is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'nofollow: {nofollow} | followed (pass equity): {followed}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`rels = rel.group(1).lower().split() if`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
	`rel else []`	
3. Check the evidence	`print(f'nofollow: {nofollow} | followed (pass equity): {followed}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rels = rel.group(1).lower().split() if rel else []`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Nofollow Versus Followed Links: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, href, rel, nofollow. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: rels = rel.group(1).lower().split() if rel else [] | Evidence: print(f'nofollow: {nofollow} | followed (pass equity): {followed}')

Reader to Writer Bridge

1. Read: explain count nofollow versus followed links in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 168.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 170 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'nofollow: {nofollow} | followed (pass equity): {followed}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count nofollow versus followed links and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The rel attribute can list several space-separated tokens, so splitting it and testing for `nofollow` membership correctly catches `rel="sponsored nofollow"` as well as a bare nofollow.

```
import re
html = ('<a href="/a" rel="nofollow">A</a>'
        '<a href="/b">B</a>'
        '<a href="https://x.com" rel="sponsored nofollow">Ad</a>'
        '<a href="/c">C</a>')
nofollow = followed = 0
for tag in re.findall(r'<a\b[^>]*>', html):
    rel = re.search(r'rel\s*=\s*"([^"]*)"', tag)
    rels = rel.group(1).lower().split() if rel else []
    if 'nofollow' in rels:
        nofollow += 1
    else:
        followed += 1
print(f'nofollow: {nofollow} | followed (pass equity): {followed}')
```

```
output
nofollow: 2 | followed (pass equity): 2
```

How it works:

1. The HTML has four anchors, two carrying a rel of nofollow including a sponsored nofollow combination.

2. Each opening anchor tag is matched, and its rel value is captured then lowercased and split into a token list.
3. When nofollow is among the tokens the nofollow counter rises, otherwise the followed counter does.
4. The script prints both totals, labelling the followed ones as passing equity.

Try it:

1. Also tally sponsored and ugc rel values as separate categories
2. List the destination of each nofollow link for review
3. Treat a missing rel as explicitly followed in the output
4. Add a `rel="ugc"` link and decide whether it counts as followed.

⚠ Common Mistakes

- Comparing the whole rel string to `'nofollow'` misses `'sponsored nofollow'`.
- A link with no rel attribute is followed by default and must land in the followed bucket.
- Case sensitivity would miss `rel="NoFollow"` without the lowercasing.

Skills gained rel parsing | nofollow audit | token split

→ Where You'll Use This: Backlink and outbound-link audits separate followed from nofollow links to estimate how much of a page's authority is actually shared with the destinations it points to.

Mental Model

A nofollow is a sealed envelope of credit marked do not deliver; followed links are the ones that actually post your endorsement onward.

Project 170 Check the Mobile Viewport Tag

The Idea Behind It

Without a responsive viewport meta tag a page renders at desktop width on phones and fails mobile-friendliness. The code captures the viewport content value and reports three states: fully OK when it sets width to device-width, present but not responsive when the tag exists without that, and missing when no viewport tag is found at all.

What You'll Build & Learn

Verify a page has a viewport meta tag that actually sets the width to the device width.

- The viewport tag enables responsive rendering
- Distinguishing present from correctly configured

- A three-branch verdict on one tag

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 169: count nofollow versus followed links. Project 171 changes the surface problem while preserving the underlying method.

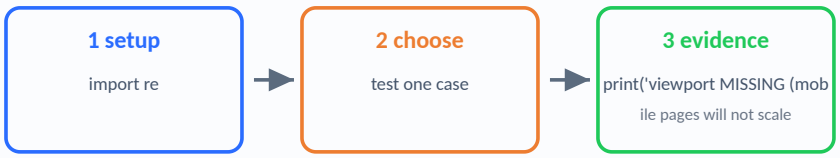
progress check

Reading target: check the mobile viewport tag is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('viewport MISSING (mobile pages will not scale)')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if m and 'width=device-width' in m.group(1):`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print('viewport MISSING (mobile pages will not scale)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m and 'width=device-width' in m.group(1):`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Check the Mobile Viewport Tag: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, head, meta, charset. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if m and 'width=device-width' in m.group(1): | Evidence: print('viewport MISSING (mobile pages will not scale)')

Reader to Writer Bridge

1. Read: explain check the mobile viewport tag in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 169.
   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 171 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('viewport MISSING (mobile pages will not scale)')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check the mobile viewport tag and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Presence of the tag is necessary but not sufficient; the content must include `width=device-width`, so the audit checks both the tag and that specific token.

```
import re
html = '<head><meta charset="utf-8"><meta name="viewport"
content="width=device-width, initial-scale=1"></head>'
m = re.search(r'<meta\s+name="viewport"\s+content="([^"]*)"', html)
if m and 'width=device-width' in m.group(1):
    print('viewport OK:', m.group(1))
elif m:
    print('viewport present but not responsive:', m.group(1))
else:
    print('viewport MISSING (mobile pages will not scale)')
```

```
output
viewport OK: width=device-width, initial-scale=1
```

How it works:

1. The head contains a charset meta and a viewport meta set to width device-width with initial scale one.
2. The regex captures the viewport content value when the tag exists.
3. An if checks that the match exists and that its content contains width equals device-width, printing viewport OK.
4. An elif handles a present but non-responsive tag, and the else reports the tag missing.

Try it:

1. Also warn when initial-scale is set to a value other than one
2. Flag a viewport that disables user zooming as an accessibility issue

3. Default the verdict appropriately when the head has no meta tags
4. Change the content to `width=980` and watch the 'not responsive' branch fire.

⚠ Common Mistakes

- A viewport tag fixed to a pixel width (e.g. `width=980`) is present but not responsive.
- Checking only that the tag exists passes a misconfigured viewport.
- An attribute-order or quoting variation can defeat the rigid regex on real pages.

Skills gained viewport check | mobile audit | branching verdict

→ Where You'll Use This: Mobile-friendliness audits flag pages without a proper viewport because they scale badly on phones, hurting rankings under mobile-first indexing and frustrating most real traffic.

Mental Model

The viewport tag is a pair of reading glasses for the phone; without it the page squints at desktop size and nobody can read the print.

Project 171 Check the HTML Lang Attribute

The Idea Behind It

The lang attribute on the html element tells browsers and engines the page's language, aiding accessibility and international targeting. The code searches the opening html tag for a lang attribute and, when found, prints the declared language code; otherwise it reports lang as missing and notes the cost to accessibility and targeting.

What You'll Build & Learn

Check whether the html element declares a lang attribute and read its value.

- Lang declares the page language
- Matching an attribute on the html element
- Two-branch present-or-missing report

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 170: check the mobile viewport tag. Project 172 changes the surface problem while preserving the underlying method.

progress check

Reading target: check the html lang attribute is the surface task; the transferable Python move is to mark setup, rule, and output before running.

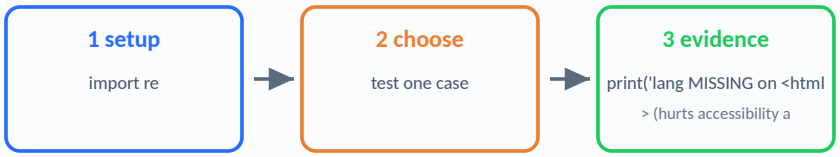
setup -> rule -> result

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('lang MISSING on <html> (hurts accessibility and targeting)')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if m:`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('lang MISSING on <html> (hurts accessibility and targeting)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m:`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Check the HTML Lang Attribute: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, lang, en, US. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if m: | Evidence: print('lang MISSING on <html> (hurts accessibility and targeting)')

🛠️ Reader to Writer Bridge

1. Read: explain check the html lang attribute in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 170.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 172 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('lang MISSING on <html> (hurts accessibility and targeting)')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of check the html lang attribute and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The regex anchors on the `<html>` opening tag and pulls its `lang` value, so a `lang` on some other element does not produce a false positive.

```
import re
html = '<!DOCTYPE html><html lang="en-US"><head><title>Hi</title></head>
</html>'
m = re.search(r'<html\b[^>]*\blang\s*=\s*"([^"]*)"', html)
if m:
    print('lang declared:', m.group(1))
else:
    print('lang MISSING on <html> (hurts accessibility and targeting)')
```

```
output
lang declared: en-US
```

How it works:

1. The HTML opens with a doctype and an html tag carrying lang equals en-US.
2. The regex anchors on the html tag and captures the lang value with a word boundary so it does not match other attributes.
3. When the search succeeds the declared code is printed.
4. When it fails the script reports the lang as missing along with the impact note.

Try it:

1. Validate that the lang value follows the language-region code format
2. Compare the declared lang against the page's actual text language
3. Detect a lang attribute placed on the body instead of html
4. Remove the lang attribute and confirm the 'MISSING' branch runs.

⚠ Common Mistakes

- Searching the whole document for `lang=` could match a `lang` on a nested element.
- An empty `lang=""` matches the regex but is still effectively undeclared.
- The `\b` before `lang` stops `xml:lang` or `data-lang` from satisfying the check.

Skills gained lang check | attribute regex | presence report

→ Where You'll Use This: Accessibility and international-SEO audits verify the lang attribute so screen readers pick the right voice and engines serve pages to the correct regional audience.

⚡ Mental Model

The lang attribute is the flag on a phrasebook's cover; without it a reader has to guess which language they are even holding.

Project 172 Find Duplicate Titles Across a Page Set

💡 The Idea Behind It

Two pages sharing one title compete with each other and confuse the results page. This project tallies titles across a small set of pages using a Counter, extracting each page's title and incrementing its count, then collects and sorts the titles that appear more than once to reveal which need to be made unique.

What You'll Build & Learn

Find titles reused across a set of pages so duplicates can be rewritten to be distinct.

- Duplicate titles cause self-competition
- Tallying occurrences with Counter
- Filtering for counts greater than one

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 171: check the html lang attribute. Project 173 changes the surface problem while preserving the underlying method.

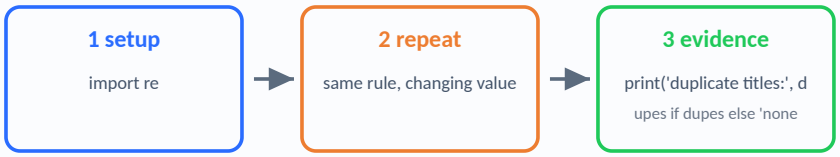
progress check

Reading target: find duplicate titles across a page set is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('duplicate titles:', dupes if dupes else 'none')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`dupes = sorted(t for t, c in titles.items() if c > 1)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('duplicate titles:', dupes if dupes else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dupes = sorted(t for t, c in titles.items() if c > 1)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Duplicate Titles Across a Page Set: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, collections, Counter, pages, shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: dupes = sorted(t for t, c in titles.items() if c > 1) | Evidence: print('duplicate titles:', dupes if dupes else 'none')

🛠️ Reader to Writer Bridge

1. Read: explain find duplicate titles across a page set in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 171.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 173 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('duplicate titles:', dupes if dupes else 'none')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find duplicate titles across a page set and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A Counter tallies how many pages share each exact title, and any title with a count above one is a duplicate worth flagging; sorting keeps the report stable.

```
import re
from collections import Counter
pages = {
    '/shoes': '<title>Running Shoes</title>',
    '/boots': '<title>Hiking Boots</title>',
```

```
    '/trail': '<title>Running Shoes</title>',
    '/socks': '<title>Wool Socks</title>',
}
titles = Counter()
for body in pages.values():
    t = re.search(r'<title>(.*?)</title>', body).group(1)
    titles[t] += 1
dupes = sorted(t for t, c in titles.items() if c > 1)
print('duplicate titles:', dupes if dupes else 'none')
```

```
output
duplicate titles: ['Running Shoes']
```

How it works:

1. A pages dictionary maps four paths to markup, two of which carry the identical title Running Shoes.
2. A Counter accumulates each extracted title, raising its count each time it recurs.
3. A comprehension keeps titles whose count exceeds one and sorts them into the dupes list.
4. The script prints the duplicate titles, or none when every title is unique.

Try it:

1. Print how many pages share each duplicated title alongside it
2. List the specific paths that carry each repeated title
3. Normalise case and whitespace before comparing titles
4. Lowercase titles before counting so case-only variants are caught.

⚠ Common Mistakes

- Comparing titles case-sensitively treats 'Running Shoes' and 'running shoes' as distinct duplicates.
- Counting tag occurrences rather than per-page titles double-counts a page with two title tags.
- Not sorting the duplicate list makes the printed order vary.

Skills gained Counter tally | duplicate detection | title audit

→ Where You'll Use This: Duplicate-title reports are standard in site crawls; auditors hand editors the list so each page earns a unique title and stops cannibalising its siblings in the rankings.

⚡ Mental Model

Duplicate titles are two shops on one street with the same name: customers and the map both get confused about which to visit.

Project 173 Check Keyword in the First Paragraph

The Idea Behind It

Engines weight the opening words of a page heavily and so do readers deciding whether to stay. The code collects all paragraphs, strips tags from the first to plain lowercased text, and checks whether the keyword appears within its first hundred characters, confirming the target phrase lands early in the intro.

What You'll Build & Learn

Confirm a target keyword appears early in a page's first paragraph.

- Early keyword placement carries weight
- Selecting the first paragraph from many
- Slicing to the first hundred characters

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 172: find duplicate titles across a page set. Project 174 changes the surface problem while preserving the underlying method.

progress check

Reading target: check keyword in the first paragraph is the surface task; the transferable Python move is to mark setup, rule, and output before running.

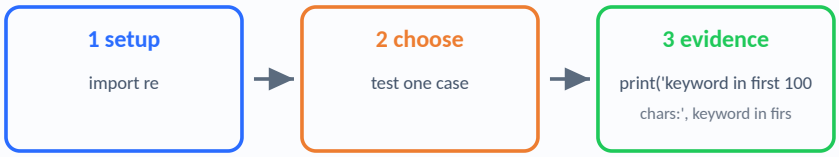
setup -> rule -> result

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as data -> transformation -> evidence, not as isolated library calls.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('keyword in first 100 chars:', keyword in first[:100])` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`first = re.sub(r'<[^>]+>', '', paras[0]).lower() if paras else ''`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. slow line
3. Check the evidence	`print('keyword in first 100 chars:', keyword in first[:100])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `first = re.sub(r'<[^>]+>', '', paras[0]).lower() if paras else ''`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Check Keyword in the First Paragraph: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, keyword, cold, brew, html. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: first = re.sub(r'<[^>]+>', '', paras[0]).lower() if paras else '' | Evidence: print('keyword in first 100 chars:', keyword in first[:100])

🛠️ Reader to Writer Bridge

1. Read: explain check keyword in the first paragraph in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 172.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 174 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('keyword in first 100 chars:', keyword in first[:100])`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check keyword in the first paragraph and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

The first `<p>` is isolated, stripped of any inner tags, and lowercased before testing whether the keyword appears within its opening characters.

```
import re
keyword = 'cold brew'
html = '<h1>Cold Brew Coffee</h1><p>Making cold brew at home is simple and
cheap.</p><p>Steep coarse grounds.</p>'
paras = re.findall(r'<p[^>]*>(.*?)</p>', html, re.S)
```

```
first = re.sub(r'<[^>]+>', '', paras[0]).lower() if paras else ''
print('first paragraph:', repr(first))
print('keyword in first 100 chars:', keyword in first[:100])
```

```
output
first paragraph: 'making cold brew at home is simple and cheap.'
keyword in first 100 chars: True
```

How it works:

1. The HTML has an H1 and two paragraphs, the first mentioning cold brew right away.
2. findall collects every paragraph's inner content, and the first is stripped of tags and lowercased.
3. An empty-paragraph guard yields an empty string if there are no paragraphs.
4. The script prints the first paragraph and reports whether the keyword sits within its first hundred characters.

Try it:

1. Make the early window configurable rather than fixed at a hundred
2. Match the keyword as separate words rather than an exact phrase
3. Report the character position where the keyword first appears
4. Move the keyword to a later paragraph and watch the early check fail.

⚠ Common Mistakes

- Treating the H1 as the first paragraph mixes the heading signal with the body signal.
- Forgetting to strip inline tags inside the paragraph can hide the keyword behind markup.
- Slicing `[:100]` on a short paragraph is harmless, but on a long one it enforces the 'early' requirement.

Skills gained intro check | string slice | keyword placement

→ Where You'll Use This: On-page optimisation expects the target phrase near the top; this check tells a writer whether the opening sentence actually leads with the topic readers and engines look for first.

Mental Model

The first paragraph is a shop window: the keyword belongs on the front display, not buried on a shelf in the back room.

Project 174 Measure Heading Keyword Coverage

The Idea Behind It

Headings structure a page's topic, so a target keyword should recur across them rather than sitting only in the H1. The code captures every heading's text, tokenises each into a lowercased word set, counts the headings

that intersect the keyword set, and reports coverage as the share of headings mentioning a keyword term.

What You'll Build & Learn

Measure what fraction of a page's headings mention any of the target keyword terms.

- Keywords should span multiple headings
- Set intersection to test for any shared term
- Coverage as a percentage of headings

🧭 Skill Roadmap

`Skill step:` moves you from recognition toward independent tracing. It builds on Project 173: check keyword in the first paragraph. Project 175 changes the surface problem while preserving the underlying method.

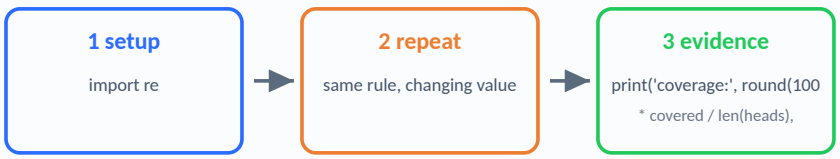
progress check

`Reading target:` measure heading keyword coverage is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as `data -> transformation -> evidence`, not as isolated library calls.
- `Writing move:` this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- `Evidence:` `print('coverage:', round(100 * covered / len(heads), 1), '%')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: `setup` creates the facts, `repeat` applies the chapter idea, and `evidence` proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if keyword & words:`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print('coverage:', round(100 * covered / len(heads), 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if keyword & words:`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure Heading Keyword Coverage: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, keyword, running, shoes, html. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: if keyword & words: | Evidence: print('coverage:', round(100 * covered / len(heads), 1), '%')

🛠️ Reader to Writer Bridge

1. Read: explain measure heading keyword coverage in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 173.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 175 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('coverage:', round(100 * covered / len(heads), 1), '%')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure heading keyword coverage and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each heading's word set is intersected with the keyword term set; a non-empty intersection means the heading is on-topic, and the share of such headings is the coverage.

```
import re
keyword = {'running', 'shoes'}
html = '<h1>Running Shoes Guide</h1><h2>Choosing Shoes</h2><h2>Care Tips</h2>
<h3>Cleaning</h3>'
heads = re.findall(r'<h[1-6][^>]*>(.*?)</h[1-6]>', html, re.S)
covered = 0
for h in heads:
    words = set(re.findall(r'[a-z]+', h.lower()))
    if keyword & words:
        covered += 1
print(f'headings: {len(heads)} | mentioning keyword: {covered}')
print('coverage:', round(100 * covered / len(heads), 1), '%')
```

```
output
headings: 4 | mentioning keyword: 2
coverage: 50.0 %
```

How it works:

1. The keyword is a set of two words and the HTML has an H1, two H2s, and an H3.
2. findall captures every heading's inner text across levels one through six.
3. Each heading's words become a lowercased set, and the intersection with the keyword set is truthy when any term matches.
4. covered counts the matching headings, and coverage is one hundred times covered over the heading total, rounded and printed.

Try it:

1. Require every keyword term rather than any single one for a hit
2. Weight the H1 and H2 matches more heavily than deeper headings
3. List which headings missed the keyword entirely
4. Require the full phrase in a heading rather than any single term.

⚠ Common Mistakes

- Mismatched open/close levels (`<h2>...</h3>`) are tolerated by `</h[1-6]>` but can capture wrong text.
- Requiring all keyword terms instead of any makes coverage unrealistically strict.
- Counting a keyword inside a longer word needs token sets, not a raw substring.

Skills gained set intersection | heading coverage | percentage

→ Where You'll Use This: Topic-coverage audits check that headings reinforce the target theme throughout the page, not just at the top, which helps engines read the page as thoroughly on-topic.

Mental Model

Headings are signposts down a trail; if only the first mentions the destination, hikers lose confidence they are still on the right path.

Project 175 Flag Image-Heavy Thin Pages

The Idea Behind It

A page that is mostly images with little surrounding text gives crawlers thin content to index. This project counts the image tags and the visible words after stripping markup, computes a words-per-image ratio guarding against division by zero, and flags the page as thin when that ratio falls below fifty.

What You'll Build & Learn

Flag image-heavy pages with too little text by computing a words-per-image ratio.

- Galleries can be content-thin for crawlers

- Guarding a ratio against zero images
- Threshold flag on words per image

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 174: measure heading keyword coverage. Project 176 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag image-heavy thin pages is the surface task; the transferable Python move is to mark setup, rule, and output before running.

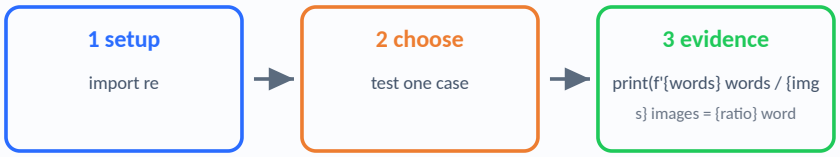
setup -> rule -> result

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{words} words / {imgs} images = {ratio} words per image -> {flag}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output.

visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

Move	Code to inspect	Why it matters
2. Apply the rule	`ratio = round(words / imgs, 1) if imgs else 0`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(f'{words} words / {imgs} images = {ratio} words per image -> {flag}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `ratio = round(words / imgs, 1) if imgs else 0`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Flag Image-Heavy Thin Pages: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, h1, Gallery, img. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: ratio = round(words / imgs, 1) if imgs else 0 | Evidence: print(f'{words} words / {imgs} images = {ratio} words per image -> {fla...

Reader to Writer Bridge

1. Read: explain flag image-heavy thin pages in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 174.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 176 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{words} words / {imgs} images = {ratio} words per image -> {flag}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag image-heavy thin pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Dividing visible word count by image count yields a words-per-image ratio; a low ratio means the images dominate and the page is light on indexable text.

```
import re
html = '<h1>Gallery</h1><img src="1.jpg" alt="a"><img src="2.jpg" alt="b">
<p>Few words here only.</p>'
imgs = len(re.findall(r'<img\b', html))
words = len(re.findall(r'[A-Za-z]+', re.sub(r'<[^>]+>', ' ', html)))
ratio = round(words / imgs, 1) if imgs else 0
flag = 'thin (image-heavy)' if ratio < 50 else 'OK'
print(f'{words} words / {imgs} images = {ratio} words per image -> {flag}')
```

```
output
5 words / 2 images = 2.5 words per image -> thin (image-heavy)
```

How it works:

1. The HTML is a gallery with a heading, two images, and one short paragraph.
2. findall counts the img tags, and stripping tags then tokenising letters counts the visible words.
3. The ratio is words over images, rounded, falling back to zero when there are no images.
4. A ratio under fifty prints the thin image-heavy flag, and the figures are reported together.

Try it:

1. Make the thinness threshold adjustable per page type
2. Count words inside image alt text toward the total

3. Report the ratio bucketed into thin, borderline, and healthy
4. Add a paragraph of copy and watch the ratio cross the threshold to OK.

⚠ Common Mistakes

- Counting words inside alt attributes would inflate the prose total and mask thinness.
- Dividing by zero on a text-only page needs the `if imgs` guard.
- Background images set via CSS are invisible to an `<img>` count and slip past this check.

Skills gained thin content | ratio guard | image count

→ **Where You'll Use This:** Product galleries and portfolio pages often trip thin-content checks; this ratio tells auditors which image-heavy pages need descriptive copy to rank for their queries.

Mental Model

Words per image is the caption-to-photo balance in a catalogue: all pictures and no copy leaves shoppers and engines without the details.

Project 176 Check Title and Description Uniqueness

The Idea Behind It

Both titles and descriptions should be distinct across a site, so counting unique values per field exposes duplicates dragging pages down. The code defines a helper that extracts either the title or description for a page, gathers all values per field, and compares the unique count to the total to declare all unique or duplicates.

What You'll Build & Learn

Check whether titles and descriptions are unique across a set of pages in one pass.

- Both title and description need uniqueness
- A helper that switches on a field name
- Comparing unique count to total count

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 175: flag image-heavy thin pages. Project 177 changes the surface problem while preserving the underlying method.

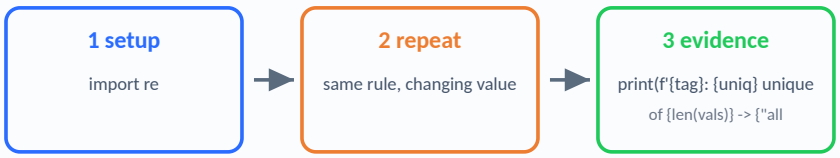
progress check

Reading target: check title and description uniqueness is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{tag}: {uniq} unique of {len(vals)} -> {"all unique" if uniq == len(vals) else "DUPLICATES"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if tag == 'title':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{tag}: {uniq} unique of {len(vals)} -> {"all unique" if uniq == len(vals) else "DUPLICATES"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if tag == 'title':`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Check Title and Description Uniqueness: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, pages, title, Trail, Shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if tag == 'title': | Evidence: print(f'{tag}: {uniq} unique of {len(vals)} -> {"all unique" if uniq ==...

Reader to Writer Bridge

1. Read: explain check title and description uniqueness in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 175.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 177 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{tag}: {uniq} unique of {len(vals)} -> {"all unique" if uniq == len(vals) else "DUPLICATES"}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check title and description uniqueness and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Collecting each field's values into a set and comparing the set size to the page count detects duplication in titles and descriptions with the same generic logic.

```
import re
pages = {
    '/a': '<title>Trail Shoes</title><meta name="description" content="Buy trail shoes.">',
    '/b': '<title>Road Shoes</title><meta name="description" content="Buy trail shoes.">',
    '/c': '<title>Trail Shoes</title><meta name="description" content="Fast road shoes.">',
}
def field(body, tag):
    if tag == 'title':
        return re.search(r'<title>(.*?)</title>', body).group(1)
    return re.search(r'name="description"\s+content="([^"]*)"', body).group(1)
for tag in ('title', 'description'):
    vals = [field(b, tag) for b in pages.values()]
    uniq = len(set(vals))
    print(f'{tag}: {uniq} unique of {len(vals)} -> {"all unique" if uniq == len(vals) else "DUPLICATES"}')
```

```
output
title: 2 unique of 3 -> DUPLICATES
description: 2 unique of 3 -> DUPLICATES
```

How it works:

1. A pages dictionary holds three pages where some titles and descriptions repeat.
2. The field helper returns the title or the description for a page depending on the tag argument.
3. For each field the values are gathered into a list and the unique count comes from wrapping them in a set.
4. When unique count equals the total the field is all unique, otherwise duplicates, printed per field.

Try it:

1. Name the specific duplicated values rather than just declaring duplicates
2. Add the H1 as a third field checked for uniqueness
3. Lowercase and trim values before testing for duplicates
4. Make all three titles distinct and watch that field report 'all unique'.

⚠ Common Mistakes

- A page missing the field would crash `.group`, so real audits need a None-safe extractor.
- Case or trailing-whitespace differences make near-duplicates look unique.
- Counting unique tags instead of per-page values misattributes duplicates on multi-tag pages.

Skills gained uniqueness check | helper function | set count

→ Where You'll Use This: Crawl reports flag duplicate titles and descriptions together; auditors use this dual check to find pages sharing snippet copy that should each be written fresh.

⚡ Mental Model

It is roll call for two attributes at once: every page should answer with its own name for both title and description, not echo a neighbour.

Project 177 Extract a Title With HTMLParser

💡 The Idea Behind It

Regex is brittle around entities and nested markup, so the standard library HTMLParser is the robust way to read element text. This project subclasses HTMLParser to track when it is inside a title and accumulate the data, correctly decoding an ampersand entity that a naive regex would have left raw, then prints the title and its length.

What You'll Build & Learn

Extract title text reliably with HTMLParser, decoding entities a regex would miss.

- HTMLParser is robust where regex is brittle
- Tracking state across start and end tags
- Accumulating text in handle_data

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 176: check title and description uniqueness. Project 178 changes the surface problem while preserving the underlying method.

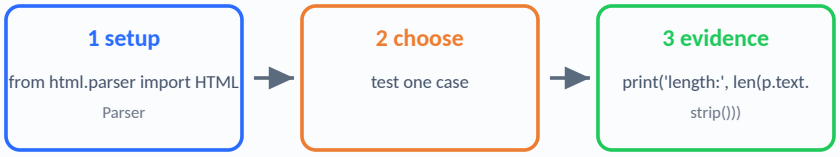
progress check

Reading target: extract a title with htmlparser is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from html.parser import HTMLParser` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is about responsibility: data and behaviour are grouped under a name, so read methods as actions that belong to that object.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('length:', len(p.text.strip()))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from html.parser import HTMLParser`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if tag == 'title':`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('length:', len(p.text.strip()))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if tag == 'title':`

Construction pattern

The construction groups state and behavior under names. Read the class/record/interface header, then ask what each method is responsible for.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

List the fields or attributes first, then read methods as behavior attached to that stored state.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Extract a Title With HTMLParser: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: html, parser, HTMLParser, TitleGrab, __init__. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; classes/records group responsibilities; functions/methods create reusable boundaries; conditions choose a route.
Data shape	Object data: state is bundled with the methods that operate on it.
Evidence	Entry: from html.parser import HTMLParser | Core: if tag == 'title': | Evidence: print('length:', len(p.text.strip()))

🛠️ Reader to Writer Bridge

1. Read: explain extract a title with htmlparser in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 176.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 178 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('length:', len(p.text.strip()))`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `from html.parser import HTMLParser` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of extract a title with htmlparser and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

HTMLParser fires `handle_data` only between the title's start and end tags, and it automatically converts `&` to `&`, giving the true rendered length a regex would miscount.

```
from html.parser import HTMLParser
class TitleGrab(HTMLParser):
    def __init__(self):
        super().__init__()
        self.intitle = False
        self.text = ''
    def handle_starttag(self, tag, attrs):
        if tag == 'title':
            self.intitle = True
    def handle_endtag(self, tag):
        if tag == 'title':
            self.intitle = False
    def handle_data(self, data):
        if self.intitle:
            self.text += data
p = TitleGrab()
p.feed('<head><title>Espresso &amp; Cold Brew Basics</title></head>')
print('title:', p.text.strip())
print('length:', len(p.text.strip()))
```

```
output
title: Espresso & Cold Brew Basics
length: 27
```

How it works:

1. A TitleGrab class subclasses HTMLParser with an intitle flag and a text accumulator.
2. handle_starttag sets intitle true on a title tag and handle_endtag clears it.
3. handle_data appends incoming text only while intitle is true, building up the title content.
4. Feeding markup with an encoded ampersand yields the decoded title, and the script prints it and its stripped length.

Try it:

1. Extend the parser to also capture the meta description content
2. Handle multiple title tags by recording only the first
3. Add handling for numeric character references beyond named entities

4. Add an `©` entity to the title and confirm it decodes to a single character.

⚠ Common Mistakes

- A regex on the raw string would count `&` as five characters, overstating the title length.
- Forgetting to flip `intitle` off in `handle_endtag` would capture text from later tags too.
- Concatenating without resetting `text` between parses would accumulate across multiple feeds.

Skills gained HTMLParser | entity decoding | stateful parsing

→ Where You'll Use This: Production audits prefer a real parser over regex for pulling text because entities, attributes with brackets, and nested tags break naive patterns on messy real-world HTML.

Mental Model

Regex reads HTML by squinting at the shapes; HTMLParser actually understands the grammar, so it never mistakes an encoded ampersand for literal text.

Project 178 Estimate Title Pixel Width

The Idea Behind It

Result pages truncate titles by pixel width rather than character count, so a short count of wide words can still be cut. This project counts the title's words, estimates its rendered width by multiplying character length by a per-character pixel figure, and warns when the estimate crosses six hundred pixels as likely to be clipped.

What You'll Build & Learn

Estimate a title's pixel width to flag whether it will be truncated in the results page.

- Truncation is by pixels not characters
- A rough pixels-per-character estimate
- Threshold verdict on estimated width

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 177: extract a title with htmlparser. Project 179 changes the surface problem while preserving the underlying method.

progress check

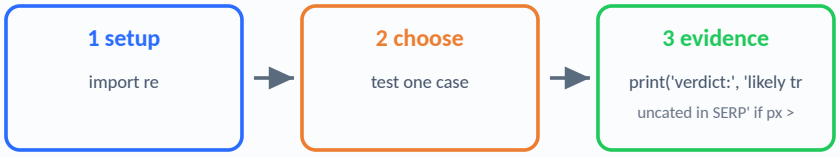
Reading target: estimate title pixel width is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits')`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Estimate Title Pixel Width: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, title, Buy, Cheap. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits') | Evidence: print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits')

🛠️ Reader to Writer Bridge

1. Read: explain estimate title pixel width in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 177.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 179 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of estimate title pixel width and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Multiplying character count by an average glyph width approximates rendered pixels; comparing against the ~600px SERP budget catches titles that fit by character count but overflow visually.

```
import re
html = '<title>Buy Cheap Running Shoes Online Now Free Shipping Best Deals 2026 Sale</title>'
title = re.search(r'<title>(.*?)</title>', html, re.S).group(1)
words = re.findall(r'[A-Za-z0-9]+', title)
px = round(len(title) * 9.5)
print(f'title: {len(words)} words, ~{px}px wide')
print('verdict:', 'likely truncated in SERP' if px > 600 else 'fits')
```

```
output
title: 12 words, ~656px wide
verdict: likely truncated in SERP
```

How it works:

1. The HTML title is a long, keyword-laden marketing string.
2. The title is captured, and findall counts its alphanumeric words.
3. Pixel width is approximated as character length times nine and a half, rounded to a whole number.
4. The script prints the word count and estimated width, then warns of likely truncation when the width exceeds six hundred.

Try it:

1. Weight wide characters like m more than narrow ones like i
2. Make the pixel-per-character and limit values configurable
3. Apply the same pixel estimate to the meta description
4. Shorten the title under 60px-equivalent and watch the verdict flip to 'fits'.

⚠ Common Mistakes

- A flat pixels-per-character figure ignores that 'W' is far wider than 'i'.
- Counting only words misses that long words still consume width.
- The 600px budget is approximate and varies by device and font.

Skills gained `pixel estimate` `truncation check` `width heuristic`

→ **Where You'll Use This:** Snippet-preview tools approximate pixel width so writers know whether a title survives the results page; this heuristic catches wide titles that pass a character-count test yet still get cut.

Mental Model

Counting characters is measuring a suitcase by its number of items; pixel width measures whether the whole thing actually fits through the overhead bin.

Project 179 Find Duplicate Alt Text on Images

The Idea Behind It

Reusing one alt string across many images helps neither accessibility nor image search. The code extracts every alt value, lowercases and trims them, tallies the non-empty ones with a Counter, and surfaces the alt strings used more than once, treating case-only differences as the same copy-pasted description.

What You'll Build & Learn

Find alt text repeated across multiple images so duplicated descriptions get rewritten.

- Duplicate alt text is unhelpful
- Case-insensitive duplicate detection
- Excluding empty alts from the tally

Skill Roadmap

`Skill step:` pushes the same idea into a less automatic situation. It builds on Project 178: estimate title pixel width. Project 180 changes the surface problem while preserving the underlying method. *progress check*

`Reading target:` find duplicate alt text on images is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- `Setup:` `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The comprehension compresses a loop into a readable noun phrase: `make this` for `each item` if `it qualifies`. Expand it mentally when it feels dense.
- `Writing move:` this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print('duplicate alt text:', dupes if dupes else 'none')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`c = Counter(a for a in alts if a)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('duplicate alt text:', dupes if dupes else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `c = Counter(a for a in alts if a)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Duplicate Alt Text on Images: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, collections, Counter, html, img. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: c = Counter(a for a in alts if a) | Evidence: print('duplicate alt text:', dupes if dupes else 'none')

🛠️ Reader to Writer Bridge

1. Read: explain find duplicate alt text on images in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 178.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 180 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('duplicate alt text:', dupes if dupes else 'none')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find duplicate alt text on images and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Lowercasing and stripping alt values before counting catches case-only variants as duplicates, and any value used more than once (ignoring empty alts) is reported.

```
import re
from collections import Counter
html = ('<img src="a.jpg" alt="Trail running shoe on rocks">'
        '<img src="b.jpg" alt="Trail running shoe on rocks">'
```

```
        '<img src="c.jpg" alt="Logo">'
        '<img src="d.jpg" alt="trail running shoe on rocks">')
alts = [m.lower().strip() for m in re.findall(r'\balt="([^"]*)"', html)]
c = Counter(a for a in alts if a)
dupes = sorted(t for t, n in c.items() if n > 1)
print('duplicate alt text:', dupes if dupes else 'none')
```

```
output
duplicate alt text: ['trail running shoe on rocks']
```

How it works:

1. Four images include three with variations of the same trail running shoe description, one differing only in case.
2. findall captures each alt value, and a comprehension lowercases and strips them into the alts list.
3. A Counter tallies only the non-empty alts, so a case difference collapses into one bucket.
4. Alts with a count above one are sorted into dupes and printed, or none when all are distinct.

Try it:

1. Report how many images share each duplicated alt string
2. Treat near-identical alts differing by one word as duplicates
3. List the src filenames that carry each repeated alt
4. Make the third repeat's alt unique and watch the duplicate vanish.

⚠ Common Mistakes

- Case-sensitive counting would miss the 'Trail…' vs 'trail…' duplicate.
- Counting empty alts as duplicates floods the report; decorative empties are filtered out.
- Not stripping whitespace lets ' Logo' and 'Logo' look distinct.

Skills gained alt duplicate | Counter tally | normalisation

→ Where You'll Use This: Image-SEO audits flag duplicated alt text because identical descriptions across a gallery dilute relevance and bore screen-reader users; auditors push for unique, specific captions.

Mental Model

Copy-pasted alt text is a museum where every painting wears the same label: visitors and the catalogue learn nothing distinct about any piece.

Project 180 Confirm the Outline Starts at H1

The Idea Behind It

A document outline should open with its single H1, so leading with an H2 signals styling chosen over meaning. The code captures the heading levels in document order, takes the first one, and reports whether the outline starts at level one, exposing a page whose first heading is not the H1.

What You'll Build & Learn

Report whether a page's first heading is its H1, as a proper outline should begin.

- Outlines should open with the H1
- Reading the first element of a list safely
- Comparing the first level to one

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 179: find duplicate alt text on images. Project 181 changes the surface problem while preserving the underlying method.

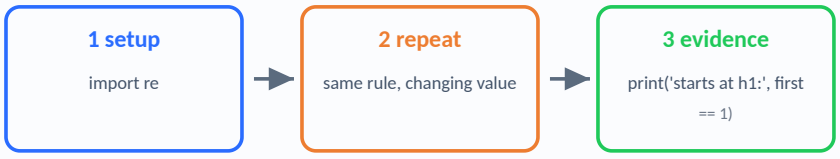
progress check

Reading target: confirm the outline starts at h1 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('starts at h1:', first == 1)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`first = levels[0] if levels else None`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('starts at h1:', first == 1)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `first = levels[0] if levels else None`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Confirm the Outline Starts at H1: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, h2, Recipes, h1. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: first = levels[0] if levels else None | Evidence: print('starts at h1:', first == 1)

Reader to Writer Bridge

1. Read: explain confirm the outline starts at h1 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 179.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 181 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('starts at h1:', first == 1)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of confirm the outline starts at h1 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Capturing heading levels in document order and inspecting the first element tells whether the outline opens at the top level or dives in mid-hierarchy.

```
import re
html = '<h2>Recipes</h2><h1>Sourdough Bread</h1><h2>Ingredients</h2>
<h3>Flour</h3>'
levels = [int(t) for t in re.findall(r'<h([1-6])', html)]
first = levels[0] if levels else None
print('heading levels in order:', levels)
print('starts at h1:', first == 1)
```

```
output
heading levels in order: [2, 1, 2, 3]
starts at h1: False
```

How it works:

1. The HTML places an H2 before the H1, an order built for appearance rather than structure.
2. findall captures each heading digit, and a comprehension turns them into an ordered levels list.
3. first takes the leading level, guarded to None when the list is empty.
4. The script prints the ordered levels and whether first equals one, reporting false here.

Try it:

1. Also confirm exactly one H1 exists anywhere in the document

2. Report at what index the first H1 actually appears
3. Warn when the page has no headings at all
4. Move the `<h1>` to the front and watch the check pass.

⚠ Common Mistakes

- Document order, not DOM nesting, is what `findall` returns; a CSS-reordered page can still mislead a reader.
- A page with no headings yields an empty list, so `first` must guard against an index error.
- Matching `<h([1-6])` without a boundary could catch an invented `<h7>`-like tag if present.

Skills gained outline start | first element | level check

→ Where You'll Use This: Heading-structure audits verify the first heading is the H1; pages that open on an H2 because it looked right in a template confuse the document outline engines build.

Mental Model

An outline starting on H2 is a book that opens on chapter two: readers and engines wonder where the missing first chapter went.

Project 181 Spot a Stuffed Meta Description

The Idea Behind It

A description that repeats one term over and over reads as spam to engines and users. The code extracts the description, removes stop words, finds the single most common remaining content word with a Counter, computes its share of the content words, and flags the description as keyword-stuffed when that share exceeds twenty percent.

What You'll Build & Learn

Detect a keyword-stuffed meta description by finding its most repeated content word's share.

- Repetition in a description reads as spam
- most_common to find the top word
- Share threshold for stuffing

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 180: confirm the outline starts at h1. Project 182 changes the surface problem while preserving the underlying method.

progress check

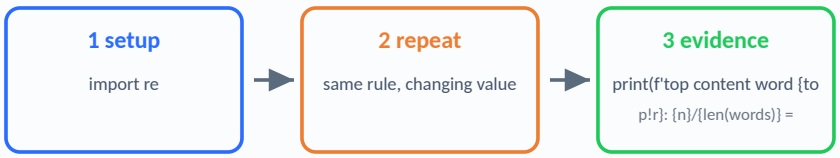
Reading target: spot a stuffed meta description is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'top content word {top!r}: {n}/{len(words)} = {ratio}% -> {flag}')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`words = [w for w in re.findall(r'[a-z]+', desc.lower()) if w not in stop]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'top content word {top!r}: {n}/{len(words)} = {ratio}% -> {flag}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `words = [w for w in re.findall(r'[a-z]+', desc.lower()) if w not in stop]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Spot a Stuffed Meta Description: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, collections, Counter, html, meta. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: words = [w for w in re.findall(r'[a-z]+', desc.lower()) if w not in sto... | Evidence: print(f'top content word {top!r}: {n}/{len(words)} = {ratio}% -> {flag}...

Reader to Writer Bridge

1. Read: explain spot a stuffed meta description in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 180.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 182 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'top content word {top!r}: {n}/{len(words)} = {ratio}% -> {flag}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of spot a stuffed meta description and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering out stop words first ensures the most-common word is a meaningful term, not 'the'; a content word claiming a large share of the text signals stuffing.

```
import re
from collections import Counter
html = '<meta name="description" content="The best running shoes, the best
deals, the best brands, the best fit, best value.">'
desc = re.search(r'content="([^"]*)"', html).group(1)
stop = {'the', 'for', 'and', 'a', 'of', 'to', 'who', 'want', 'on'}
words = [w for w in re.findall(r'[a-z]+', desc.lower()) if w not in stop]
top, n = Counter(words).most_common(1)[0]
ratio = round(100 * n / len(words), 1)
flag = 'keyword-stuffed' if ratio > 20 else 'OK'
print(f'top content word {top!r}: {n}/{len(words)} = {ratio}% -> {flag}')
```

```
output
top content word 'best': 5/11 = 45.5% -> keyword-stuffed
```

How it works:

1. The description hammers the word best across a comma-laden sentence.
2. After capturing the content, a comprehension keeps lowercased words that are not in the stop set.
3. Counter most_common returns the top content word and its count, and the ratio is its share of all content words, rounded.
4. A ratio above twenty percent prints the keyword-stuffed flag along with the figures.

Try it:

1. Apply the same stuffing check to the page title
2. Make the stuffing threshold percentage configurable
3. Report the top three content words and their shares
4. Vary the repeated word with synonyms and watch the ratio fall to OK.

⚠ Common Mistakes

- Skipping the stop-word filter makes 'the' the top word and hides the real stuffing.
- A threshold set too high never flags genuine over-repetition.
- `most_common(1)` on an all-stop-word description would crash on an empty list.

Skills gained stuffing detection | Counter most_common | stop filtering

→ Where You'll Use This: Spam and quality audits flag stuffed snippets so writers rework descriptions into natural sentences, since overt repetition can suppress a snippet or read as manipulative to engines.

Mental Model

A stuffed description is a salesperson who only shouts best, best, best: the word loses meaning and the pitch starts to sound desperate.

Project 182 Compare Title and H1 Word Overlap

The Idea Behind It

The title and the H1 should describe the same thing, so a wide gap between them is a relevance smell. This project lowercases both, reduces each to a set of words, and intersects them to find the shared words, reporting an empty overlap as a sign the two signals are disconnected rather than reinforcing.

What You'll Build & Learn

Measure the shared words between a page's title and H1 to test whether they agree.

- Title and H1 should reinforce each other
- Word sets and their intersection
- Empty overlap as a warning sign

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 181: spot a stuffed meta description. Project 183 changes the surface problem while preserving the underlying method.

progress check

Reading target: compare title and h1 word overlap is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

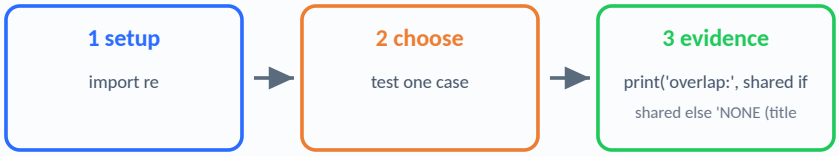
Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('overlap:', shared if shared else 'NONE (title and h1 disconnected)')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`print('overlap:', shared if shared else 'NONE (title and h1 disconnected)')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('overlap:', shared if shared else 'NONE (title and h1 disconnected)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('overlap:', shared if shared else 'NONE (title and h1 disconnected)')`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Compare Title and H1 Word Overlap: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, title, Home, h1. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print('overlap:', shared if shared else 'NONE (title and h1 disconnecte... | Evidence: print('overlap:', shared if shared else 'NONE (title and h1 disconnecte...

🛠️ Reader to Writer Bridge

1. Read: explain compare title and h1 word overlap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 181.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 183 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('overlap:', shared if shared else 'NONE (title and h1 disconnected)')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of compare title and h1 word overlap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Reducing each element to a lowercase word set and intersecting them surfaces shared vocabulary; an empty intersection means the title and H1 are sending different topic signals.

```
import re
html = '<title>Home</title><h1>Welcome to Acme Running Co</h1>'
title = re.search(r'<title>(.*?)</title>', html, re.S).group(1).lower()
h1 = re.search(r'<h1[^>]*>(.*?)</h1>', html, re.S).group(1).lower()
tset = set(re.findall(r'[a-z]+', title))
hset = set(re.findall(r'[a-z]+', h1))
shared = sorted(tset & hset)
print('title words:', sorted(tset))
print('h1 words:', sorted(hset))
print('overlap:', shared if shared else 'NONE (title and h1 disconnected)')
```

```
output
title words: ['home']
h1 words: ['acme', 'co', 'running', 'to', 'welcome']
overlap: NONE (title and h1 disconnected)
```

How it works:

1. The HTML pairs a vague title Home with a specific H1 about Acme Running Co.
2. Both are captured and lowercased, then findall reduces each to a set of words.
3. The intersection of the two sets is sorted into the shared list.
4. The script prints each word set and the overlap, reporting none here since the two share no words.

Try it:

1. Report the overlap as a percentage of the title's words
2. Ignore stop words before computing the shared set
3. Extend the comparison to include the first H2 as well
4. Rewrite the title to include 'Acme Running' and watch the overlap appear.

⚠ Common Mistakes

- Comparing raw strings instead of word sets reports a mismatch even for reordered same-word phrases.
- Not lowercasing makes 'Home' and 'home' fail to overlap.
- Stop words inflate apparent overlap, so a shared 'the' can mask a real topic gap.

Skills gained set overlap title-h1 match relevance check

→ Where You'll Use This: Relevance audits compare title and H1 wording because a disconnect suggests a templated or careless title; aligning them helps both signals point engines at the same topic.

Mental Model

Title and H1 are two witnesses to the same event; when their stories share no details, you suspect one of them was not really there.

Project 183 Score a Page Against an On-Page Checklist

The Idea Behind It

Real audits roll many checks into one score, and this one tallies four essentials in a single pass. The code tests for a non-empty title, a non-empty description, a canonical link, and a viewport tag, stores each result as a boolean in a dictionary, sums them for a score, and prints present or missing per check.

What You'll Build & Learn

Score one page against four on-page essentials and print a pass-fail tally.

- Bundling checks into one score
- Storing boolean results in a dictionary
- Summing booleans to get a total

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 182: compare title and h1 word overlap. Project 184 changes the surface problem while preserving the underlying method.

progress check

Reading target: score a page against an on-page checklist is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print(f'score: {score}/{len(required)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`print(f'{k}: {"present" if required[k] else "MISSING"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'score: {score}/{len(required)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{k}: {"present" if required[k] else "MISSING"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score a Page Against an On-Page Checklist: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, head, title, Pricing. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print(f'{k}: {"present" if required[k] else "MISSING"}') | Evidence: print(f'score: {score}/{len(required)}')

🛠️ Reader to Writer Bridge

1. Read: explain score a page against an on-page checklist in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 182.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 184 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'score: {score}/{len(required)}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score a page against an on-page checklist and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each check is a boolean regex search, and the `\S` requirement on title and description ensures a present-but-empty tag does not count; summing the booleans gives the score.

```
import re
html = '<head><title>Pricing Plans</title><meta name="description"
content="Compare our plans."></head>'
required = {
    'title': bool(re.search(r'<title>.*?\S.*?</title>', html, re.S)),
    'description': bool(re.search(r'name="description"\s+content="[^"]*\S',
html)),
    'canonical': bool(re.search(r'rel="canonical"', html)),
    'viewport': bool(re.search(r'name="viewport"', html)),
}
score = sum(required.values())
for k in sorted(required):
    print(f'{k}: {"present" if required[k] else "MISSING"}')
print(f'score: {score}/{len(required)}')
```

```
output
canonical: MISSING
description: present
title: present
viewport: MISSING
score: 2/4
```

How it works:

1. The HTML head supplies a title and description but no canonical or viewport.
2. A required dictionary maps each check name to a boolean from a regex, where the title and description patterns also require a non-whitespace character.
3. score sums the dictionary values, since True counts as one.
4. A sorted loop prints present or missing per check, then the script prints the score out of four.

Try it:

1. Add an H1-present check as a fifth essential
2. Weight the title check more heavily than the others
3. Print only the checks that failed for a terse report
4. Add a canonical and viewport tag and watch the score climb to 4/4.

⚠ Common Mistakes

- A title regex without `\S` would count an empty `<title></title>` as present.
- Summing booleans relies on True equalling 1; mixing in None values would break the count.
- Iterating the dict unsorted makes the per-check lines print in nondeterministic order.

Skills gained scorecard boolean sum checklist audit

→ Where You'll Use This: Audit dashboards condense many on-page checks into a single page score; this pattern is the seed of any report card that ranks pages by how many essentials they pass.

Mental Model

It is a vehicle inspection: four boxes to tick, and the sticker shows how many passed so you know what still needs a repair.

Project 184 Score Keyword Placement Across Elements

The Idea Behind It

Where a keyword appears matters as much as whether it appears, so this scores its presence across four prime spots. The code captures the title, H1, an H2, and the first paragraph into a dictionary, lists the keys whose text contains the keyword, and reports both the placements found and a score out of four.

What You'll Build & Learn

Score a keyword's presence across the title, H1, an H2, and the first paragraph.

- Placement spread strengthens a keyword
- Building a dictionary of page regions
- Listing keys whose value matches

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 183: score a page against an on-page checklist. Project 185 changes the surface problem while preserving the underlying method.

progress check

Reading target: score keyword placement across elements is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print('placement score:', len(hits), 'of', len(spots))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`hits = [k for k, v in spots.items() if keyword in v.lower()]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('placement score:', len(hits), 'of', len(spots))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `hits = [k for k, v in spots.items() if keyword in v.lower()]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score Keyword Placement Across Elements: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, keyword, standing, desk, html. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: hits = [k for k, v in spots.items() if keyword in v.lower()] | Evidence: print('placement score:', len(hits), 'of', len(spots))

🛠️ Reader to Writer Bridge

1. Read: explain score keyword placement across elements in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 183.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 185 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('placement score:', len(hits), 'of', len(spots))`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score keyword placement across elements and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each element is extracted into a dict and tested for the lowercased keyword; the count of element types containing it is the placement score across the most weighted on-page spots.

```
import re
keyword = 'standing desk'
html = '<title>Adjustable Standing Desk</title><h1>The Best Standing Desk</h1>
<h2>Standing Desk Setup</h2><p>A standing desk improves posture.</p>'
spots = {
    'title': re.search(r'<title>(.*?)</title>', html, re.S).group(1),
    'h1': re.search(r'<h1[^>]*>(.*?)</h1>', html, re.S).group(1),
    'h2': re.search(r'<h2[^>]*>(.*?)</h2>', html, re.S).group(1),
    'first_p': re.search(r'<p[^>]*>(.*?)</p>', html, re.S).group(1),
}
hits = [k for k, v in spots.items() if keyword in v.lower()]
print('keyword placement:', sorted(hits))
print('placement score:', len(hits), 'of', len(spots))
```

```
output
keyword placement: ['first_p', 'h1', 'h2', 'title']
placement score: 4 of 4
```

How it works:

1. The HTML mentions standing desk in the title, H1, H2, and the opening paragraph.
2. A spots dictionary captures the inner text of each of the four regions with its own regex.
3. A comprehension collects the keys whose lowercased text contains the keyword into the hits list.
4. The script prints the sorted placements and a score of how many of the four spots matched.

Try it:

1. Add the meta description as a fifth scored placement
2. Weight the title and H1 hits above the H2 and paragraph
3. Match keyword words individually rather than the whole phrase
4. Drop the keyword from the H2 and watch the score fall to 3 of 4.

⚠ Common Mistakes

- Each extraction assumes the tag exists; a missing H2 would crash on `.group(1)`.
- Only the first H2 is checked, so a keyword in a later H2 is missed.
- A substring match counts the keyword even inside an unrelated longer phrase.

Skills gained placement score | region dict | keyword spread

→ Where You'll Use This: On-page scoring tools reward keyword spread across key elements; this placement score gives an optimiser a concrete number for how well a page surfaces its target term.

⚡ Mental Model

Keyword placement is planting the same flag at four checkpoints: scattering it across the map proves you really hold the territory.

Project 185 Infer Search Intent From the Title

💡 The Idea Behind It

Titles leak the searcher intent they target, so cue words classify a page's likely intent from its title alone. The code defines word lists for informational, transactional, and navigational intents, scores each by summing how often its cue words occur in the lowercased title, and picks the highest-scoring intent as the inferred one.

What You'll Build & Learn

Infer a page's likely search intent by scoring intent cue words in its title.

- Titles signal the searcher's intent
- Scoring categories by cue word counts
- Choosing the max-scoring category

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 184: score keyword placement across elements. Project 186 changes the surface problem while preserving the underlying method.

progress check

Reading target: infer search intent from the title is the surface task; the transferable Python move is to mark setup, rule, and output before running.

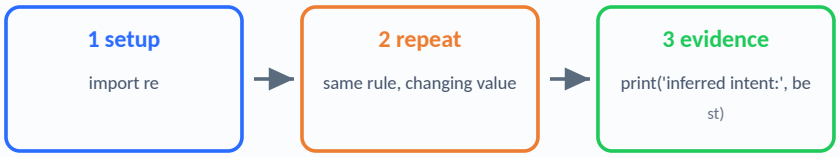
setup -> rule -> result

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('inferred intent:', best)` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`scores = {kind: sum(title.count(w) for w in words) for kind, words in intents.items()}`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('inferred intent:', best)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `scores = {kind: sum(title.count(w) for w in words) for kind, words in intents.items()}`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Infer Search Intent From the Title: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, intents, informational, how, what. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: scores = {kind: sum(title.count(w) for w in words) for kind, words in i... | Evidence: print('inferred intent:', best)

🛠️ Reader to Writer Bridge

1. Read: explain infer search intent from the title in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 184.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 186 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('inferred intent:', best)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of infer search intent from the title and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Each intent category accumulates a score from its cue words, and the highest-scoring category (with sorted keys to break ties deterministically) is the inferred intent.

```
import re
intents = {
    'informational': ('how', 'what', 'guide', 'tips', 'why'),
    'transactional': ('buy', 'price', 'cheap', 'order', 'shop'),
    'navigational': ('login', 'official', 'website'),
}
```

```
title = re.search(r'<title>(.*?)</title>', '<title>How to Brew Cold Brew: A
Beginner Guide</title>', re.S).group(1).lower()
scores = {kind: sum(title.count(w) for w in words) for kind, words in
intents.items()}
best = max(sorted(scores), key=lambda k: scores[k])
print('signals:', {k: scores[k] for k in sorted(scores)})
print('inferred intent:', best)
```

```
output
signals: {'informational': 2, 'navigational': 0, 'transactional': 0}
inferred intent: informational
```

How it works:

1. An intents dictionary maps three intent types to tuples of cue words.
2. The title how to brew cold brew a beginner guide is captured and lowercased inline.
3. A dict comprehension scores each intent by summing the title's counts of its cue words.
4. max over the sorted keys picks the top intent, and the script prints the per-intent signals and the inferred intent.

Try it:

1. Add a commercial-investigation intent with cues like best and review
2. Break ties by preferring the intent with the rarest matched cue
3. Score the H1 alongside the title for a stronger signal
4. Swap the title to 'Buy Cold Brew Maker' and watch intent flip to transactional.

⚠ Common Mistakes

- `title.count('how')` also fires inside 'however', so cue words can over-match.
- Calling `max` over the unsorted dict makes ties resolve unpredictably.
- A title with no cue words returns the first category by sort order, not a real 'unknown'.

Skills gained intent inference | cue scoring | max selection

→ Where You'll Use This: Intent classification helps match content to the queries it should target; auditors use title cues to spot pages aimed at the wrong stage of a searcher's journey.

Mental Model

Intent cues are tells in a card game: how to whispers a curious browser while buy now shows a wallet already out.

Project 186 Check Title Intent Against Page Intent

The Idea Behind It

A product page should read transactional, not informational, so comparing signalled intent against intended intent flags a mismatch. The code scores informational and transactional cue words in the title, picks the detected intent, compares it to the declared page intent, and reports whether the two match or clash.

What You'll Build & Learn

Compare the intent a title signals against the intent the page targets and flag mismatches.

- Title intent should match page intent
- Selecting a detected category by max score
- Equality test for a mismatch verdict

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 185: infer search intent from the title. Project 187 changes the surface problem while preserving the underlying method.

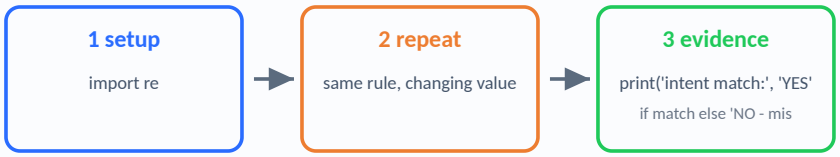
progress check

Reading target: check title intent against page intent is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('intent match:', 'YES' if match else 'NO – mismatch')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`print('intent match:', 'YES' if match else 'NO – mismatch')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('intent match:', 'YES' if match else 'NO – mismatch')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('intent match:', 'YES' if match else 'NO – mismatch')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Title Intent Against Page Intent: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, page_intent, transactional, title, search. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: import re | Core: print('intent match:', 'YES' if match else 'NO - mismatch') | Evidence: print('intent match:', 'YES' if match else 'NO - mismatch')

🛠️ Reader to Writer Bridge

1. Read: explain check title intent against page intent in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 185.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 187 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('intent match:', 'YES' if match else 'NO – mismatch')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check title intent against page intent and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The detected intent comes from the highest-scoring cue category, and comparing it to the page's declared target reveals when the title chases a different intent than the page serves.

```
import re
page_intent = 'transactional'
title = re.search(r'<title>(.*?)</title>', '<title>How to Choose Running Shoes:
Buying Guide</title>', re.S).group(1).lower()
intent_words = {'informational': ('how', 'guide', 'what', 'tips'),
                'transactional': ('buy', 'shop', 'price', 'order')}
detected = max(sorted(intent_words), key=lambda k: sum(title.count(w) for w in
intent_words[k]))
match = detected == page_intent
print(f'page targets: {page_intent}')
```

```
print(f'title signals: {detected}')
print('intent match:', 'YES' if match else 'NO — mismatch')
```

```
output
page targets: transactional
title signals: informational
intent match: NO — mismatch
```

How it works:

1. The page intent is declared transactional but the title reads how to choose running shoes buying guide.
2. An intent words dictionary maps two intents to cue tuples, and the title is captured and lowercased.
3. max over the sorted keys picks the detected intent by summed cue counts, landing on informational.
4. match compares detected to the declared intent, and the script prints both and a yes or no mismatch verdict.

Try it:

1. Add navigational and commercial intents to the comparison
2. Report by how many cue words the detected intent led
3. Let the page intent be inferred from the URL pattern instead
4. Rewrite the title as 'Buy Running Shoes Online' and watch the match become YES.

⚠ Common Mistakes

- 'Buying Guide' scores informational here because 'guide' outweighs no transactional cue word.
- A title with zero cues defaults to the first sorted category, faking a confident detection.
- Substring counting lets 'buying' miss the cue word 'buy' if you required whole words.

Skills gained intent mismatch | max scoring | equality check

→ Where You'll Use This: Content-to-intent audits catch pages whose copy fights their purpose, like a shop page written as a guide; aligning the two improves both rankings and conversion.

Mental Model

It is matching a label to its contents: a jar marked jam that smells of pickles tells you something got mixed up on the line.

Project 187 Detect Structured Data Types

The Idea Behind It

JSON-LD structured data unlocks rich results like FAQ panels and review stars. The code finds every JSON-LD script block, joins their contents, extracts each declared schema type from the at-type fields, and reports how

many blocks exist and which schema types the page declares, deduplicated and sorted.

What You'll Build & Learn

Find embedded JSON-LD blocks and report which schema types the page declares.

- JSON-LD enables rich results
- Matching script blocks by their type attribute
- Pulling the at-type values from the payload

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 186: check title intent against page intent. Project 188 changes the surface problem while preserving the underlying method.

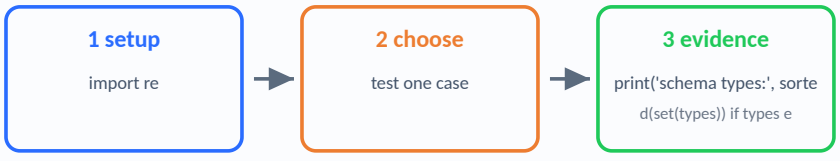
progress check

Reading target: detect structured data types is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print('schema types:', sorted(set(types)) if types else 'none')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`print('schema types:', sorted(set(types)) if types else 'none')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('schema types:', sorted(set(types)) if types else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('schema types:', sorted(set(types)) if types else 'none')`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Detect Structured Data Types: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, head, title, script. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.

Layer	What to read
Evidence	Entry: import re | Core: print('schema types:', sorted(set(types)) if types else 'none') | Evidence: print('schema types:', sorted(set(types)) if types else 'none')

🛠️ Reader to Writer Bridge

1. Read: explain detect structured data types in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 186.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 188 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('schema types:', sorted(set(types)) if types else 'none')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect structured data types and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The script-tag regex isolates JSON-LD payloads, and a second regex harvests every `@type` value so the audit lists the structured-data kinds present without fully parsing the JSON.

```
import re
html = '<head><title>FAQ</title><script type="application/ld+json">
{"@type":"FAQPage"}</script></head>'
blocks = re.findall(r'<script[^>]*application/ld\+json[^>]*>(.*?)</script>',
html, re.S)
types = re.findall(r'"@type"\s*:\s*"([^"]+)"', ' '.join(blocks))
print('JSON-LD blocks:', len(blocks))
print('schema types:', sorted(set(types)) if types else 'none')
```

```
output
JSON-LD blocks: 1
schema types: ['FAQPage']
```

How it works:

1. The head contains one JSON-LD script block declaring an FAQPage type.
2. findall captures the inner content of every script tag whose type is application of the ld plus json kind.
3. The blocks are joined and a second findall pulls each at-type value from them.
4. The script prints the block count and the sorted set of schema types, or none when no types are found.

Try it:

1. Validate that each block is parseable JSON rather than only matching it
2. Detect microdata itemtype attributes in addition to JSON-LD
3. Warn when an expected schema type for the page is absent
4. Add a second JSON-LD block with a `Product` type and watch both types list.

⚠ Common Mistakes

- Escaping the literal `+` in `ld+json` is required, or the regex treats it as a quantifier.
- Reading `@type` by regex misses types written across multiple lines or nested oddly.
- Microdata or RDFa structured data uses different markup and is invisible to this JSON-LD-only check.

Skills gained JSON-LD detection | schema types | script matching

→ Where You'll Use This: Structured-data audits verify pages carry the schema needed for rich results; this scan tells which types a page declares so you can confirm coverage for FAQ, product, or article markup.

⚡ Mental Model

Structured data is a backstage cue card for engines: it spells out this is an FAQ so they can stage the fancy result instead of a plain link.

Project 188 Flag Inconsistent Anchor Text to One URL

💡 The Idea Behind It

Linking to one page with wildly different anchor text muddies the relevance signal that text sends. This project groups anchors by their destination using a defaultdict of sets, collects every distinct trimmed label per href, and flags any URL described with more than one label as inconsistent.

What You'll Build & Learn

Flag any URL on a page that is linked with more than one different anchor text.

- Consistent anchor text sharpens relevance
- Grouping with a defaultdict of sets

- Flagging destinations with multiple labels

Skill Roadmap

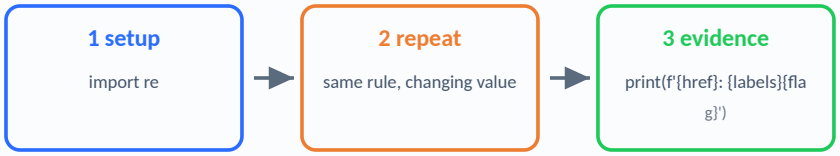
Skill step: pushes the same idea into a less automatic situation. It builds on Project 187: detect structured data types. Project 189 changes the surface problem while preserving the underlying method. *progress check*

Reading target: flag inconsistent anchor text to one url is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'{href}: {labels}{flag}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`flag = ' <- inconsistent' if len(labels) > 1 else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{href}: {labels}{flag}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = ' <- inconsistent' if len(labels) > 1 else ''`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Inconsistent Anchor Text to One URL: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, collections, defaultdict, html, href. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: flag = ' <- inconsistent' if len(labels) > 1 else '' | Evidence: print(f'{href}: {labels} {flag}')

Reader to Writer Bridge

1. Read: explain flag inconsistent anchor text to one url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 187.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 189 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{href}: {labels} {flag}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag inconsistent anchor text to one url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Mapping each destination href to the set of anchor texts that point at it exposes URLs described more than one way; a set size above one is the inconsistency flag.

```
import re
from collections import defaultdict
html = ('<a href="/home">Home</a><a href="/blog">Blog</a>'
        '<a href="/blog">Blog</a><a href="/about">About</a>'
        '<a href="/blog">Read the blog</a>')
texts = defaultdict(set)
for href, txt in re.findall(r'<a[^>]*href="([^"]*)"[^>]*>(.*?)</a>', html,
re.S):
    texts[href].add(txt.strip())
for href in sorted(texts):
    labels = sorted(texts[href])
    flag = ' <- inconsistent' if len(labels) > 1 else ''
    print(f'{href}: {labels}{flag}')
```

```
output
/about: ['About']
/blog: ['Blog', 'Read the blog'] <- inconsistent
/home: ['Home']
```

How it works:

1. The HTML links to the blog twice as Blog and once as Read the blog, plus single links to home and about.
2. A defaultdict of sets maps each href to the distinct trimmed anchor texts pointing at it.
3. findall yields each href and its text, which is stripped and added to the destination's set.
4. Sorted destinations print their labels, appending an inconsistent flag when a URL has more than one label.

Try it:

1. Treat case-only differences in labels as the same anchor text
2. Report the most common label per URL as the suggested standard
3. Count how many distinct labels each inconsistent URL carries

4. Make every link to /blog use 'Blog' and watch the inconsistency clear.

⚠ Common Mistakes

- Using a list instead of a set would report a repeated identical anchor as 'inconsistent'.
- Not stripping anchor text makes ' Blog' and 'Blog' look like two different labels.
- The non-greedy `(.*?)` is needed so each anchor is captured separately, not run together.

Skills gained anchor consistency | defaultdict grouping | label set

→ Where You'll Use This: Internal-link audits flag inconsistent anchors so a destination receives a coherent relevance signal; scattering different labels at one URL weakens the topic it should rank for.

Mental Model

Anchor text is how friends describe a restaurant to others; if everyone calls it something different, no one is sure they all mean the same place.

Project 189 Check Keyword Position Within the Title

The Idea Behind It

A keyword near the front of a title carries more weight and survives truncation, so locating its position tells whether it is front-loaded or buried. The code finds the keyword's character index in the lowercased title, computes that as a percentage of the title length, and labels it front-loaded, early, or buried.

What You'll Build & Learn

Find where a keyword sits within a title and judge whether it is front-loaded or buried.

- Front-loaded keywords carry more weight
- Finding a substring position with find
- Mapping position percentage to a label

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 188: flag inconsistent anchor text to one url. Project 190 changes the surface problem while preserving the underlying method.

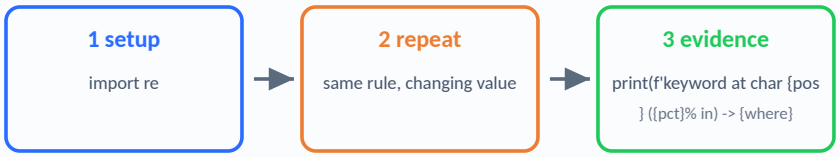
progress check

Reading target: check keyword position within the title is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
- Evidence: `print(f'keyword at char {pos} ({pct}% in) -> {where}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: `setup` creates the facts, `repeat` applies the chapter idea, and `evidence` proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`if pos < 0:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'keyword at char {pos} ({pct}% in) -> {where}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if pos < 0:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Keyword Position Within the Title: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, keyword, yoga, mat, title. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if pos < 0: | Evidence: print(f'keyword at char {pos} ({pct}% in) -> {where}')

🛠️ Reader to Writer Bridge

1. Read: explain check keyword position within the title in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 188.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 190 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'keyword at char {pos} ({pct}% in) -> {where}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of check keyword position within the title and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`str.find` returns the keyword's first index, and converting it to a percentage of title length buckets the placement into front-loaded, early, or buried.

```
import re
keyword = 'yoga mat'
title = re.search(r'<title>(.*?)</title>', '<title>Eco Yoga Mat: Non-Slip Cork Mat for Hot Yoga</title>', re.S).group(1).lower()
pos = title.find(keyword)
if pos < 0:
    print('keyword not in title')
else:
    pct = round(100 * pos / len(title), 1)
    where = 'front-loaded' if pos == 0 else ('early' if pct < 30 else 'buried')
    print(f'keyword at char {pos} ({pct}% in) -> {where}')
```

```
output
keyword at char 4 (9.1% in) -> early
```

How it works:

1. The title Eco Yoga Mat is captured, lowercased, and searched for the keyword yoga mat.
2. find returns the character index, with a negative result printing a not-in-title message.
3. The position is converted to a percentage of the title length, rounded.
4. A position of zero is front-loaded, under thirty percent is early, and beyond that is buried, printed with the figures.

Try it:

1. Make the early versus buried percentage cutoff configurable
2. Report the position of each word of a multi-word keyword
3. Apply the same position check to the H1
4. Move the keyword to the very start and watch it read 'front-loaded'.

⚠ Common Mistakes

- `find` returns -1 when absent, which must be handled before treating it as a position.
- Lowercasing only the title (not the keyword) would miss a capitalised keyword constant.
- Position as a percentage depends on title length, so a long title pushes the 'buried' threshold.

Skills gained keyword position str.find position labelling

→ Where You'll Use This: Title-optimisation tools report keyword position because front-loading the target phrase helps it survive truncation and signals priority; auditors flag buried keywords for rewriting.

Mental Model

Keyword position is seating at a dinner: the head of the table commands attention, while the far end is easy to overlook.

Project 190 Run the Full On-Page Audit Report

The Idea Behind It

This capstone ties the chapter together by running the core on-page checks from one HTML string and printing a single scorecard. The code tests title length, description length, a single H1, a canonical, a lang attribute, and that all images carry alt, appending each result, summing the passes, and printing a checkbox line per check.

What You'll Build & Learn

Run the chapter's core on-page checks on one page and print a combined pass-fail scorecard.

- Combining many checks into a report
- Appending result tuples to a list
- Using all to test every image for alt

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 189: check keyword position within the title. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run the full on-page audit report is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.

- Evidence: `print(f'on-page score: {passed}/{len(checks)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
2. Apply the rule	`print(f'[{"x" if ok else " "}] {name}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'on-page score: {passed}/{len(checks)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: On-Page Signals and the HTML Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'[{"x" if ok else " "}] {name}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run the Full On-Page Audit Report: connect the surface task to the chapter skill, audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet.
Names	Watch these names first: re, html, lang, en, head. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print(f'[{"x" if ok else " "}] {name}') | Evidence: print(f'on-page score: {passed}/{len(checks)}')

🛠️ Reader to Writer Bridge

1. Read: explain run the full on-page audit report in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 189.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'on-page score: {passed}/{len(checks)}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run the full on-page audit report and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Each on-page rule from the chapter is reduced to a boolean and collected in order, so summing the booleans yields a single scorecard that mirrors a real audit summary.

```
import re
html = ('<html lang="en"><head><title>Trail Running Shoes Guide for
Beginners</title>'
        '<meta name="description" content="A clear guide to picking your first
pair of trail running shoes.">'
        '<link rel="canonical" href="https://example.com/guide"></head>'
        '<body><h1>Trail Running Shoes</h1><img src="a.jpg" alt="Shoe"><img
src="b.jpg"></body></html>')
checks = []
t = re.search(r'<title>(.*?)</title>', html, re.S).group(1)
checks.append(('title 50-60', 50 <= len(t) <= 60))
d = re.search(r'description"\s+content="([^"]*)"', html).group(1)
checks.append(('desc <=155', len(d) <= 155))
checks.append(('exactly one h1', len(re.findall(r'<h1', html)) == 1))
checks.append(('canonical', bool(re.search(r'rel="canonical"', html))))
checks.append(('lang set', bool(re.search(r'<html[^>]*lang=', html))))
imgs = re.findall(r'<img\b[^>]*>', html)
checks.append(('all imgs alt', all(re.search(r'\balt=', i) for i in imgs)))
passed = sum(ok for _, ok in checks)
for name, ok in checks:
    print(f'[{"x" if ok else " "}] {name}')
print(f'on-page score: {passed}/{len(checks)}')
```

```
output
[ ] title 50-60
[x] desc <=155
[x] exactly one h1
[x] canonical
[x] lang set
[ ] all imgs alt
on-page score: 4/6
```

How it works:

1. A full HTML string carries a lang, title, description, canonical, an H1, and two images, one without alt.
2. Each check appends a name and boolean tuple, covering title length, description length, single H1, canonical, lang, and image alts.
3. The image check uses all over the img tags so it fails the moment any image lacks alt.
4. passed sums the boolean results, each check prints with an x or blank box, and the final score reads four of six.

Try it:

1. Add checks for a viewport meta and an Open Graph title
2. Print a remediation hint next to each failing check
3. Run the whole scorecard over a dictionary of several pages

4. Lengthen the title into the 50-60 window and watch the score rise.

⚠ Common Mistakes

- A 39-character title fails the 50-60 window, so this sample scores 4 of 6, not a perfect 6.
- Counting `<h1` opening tags assumes well-formed markup; an unclosed tag would still count.
- The second image has no alt, so the 'all imgs alt' check fails as intended.

Skills gained full audit | all aggregation | scorecard report

→ Where You'll Use This: This is the shape of a real on-page audit: many independent checks feeding one page score, which is exactly how crawlers and dashboards summarise a page's health at a glance.

Mental Model

The capstone is a full health checkup: pulse, blood pressure, and the rest measured in one visit, summed into a single chart of what is fine and what needs care.

✅ Chapter Progress Checkpoint

What this chapter added: after 40 mini-projects, On-Page Signals and the HTML Audit should feel like a reusable Python move: audit titles, meta descriptions, header hierarchy, alt text and intent match from an html snippet. *chapter checkpoint*

1 Read
Open any project from Projects 151-190 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.
transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 6 (Keyword Research and Intent as Data) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

9 questions · 4 categories · graded

3 Recall | 2 Meaning | 2 Structure | 2 Transfer

5.1.Q1 Recall MEDIUM Turn back to Project 174 (Measure Heading Keyword Coverage). Without running it, what exactly does it print?

5.1.Q2 Meaning MEDIUM Turn back to Project 160 (Verify a Canonical Link Is Present). Without running it, what exactly does it print?

5.1.Q3 Structure MEDIUM Turn back to Project 185 (Infer Search Intent From the Title). Without running it, what exactly does it print?

5.1.Q4 Transfer MEDIUM Turn back to Project 175 (Flag Image-Heavy Thin Pages). Without running it, what exactly does it print?

5.1.Q5 Recall EASY What is the conventionally healthy length range for a **<title>** tag, and why?

- A 10-20 characters, for brevity
- B 50-60 characters, so search results do not truncate it
- C Exactly 100 characters
- D There is no limit

5.1.Q6 Meaning EASY A page has two **<h1>** tags. Why is that an on-page fault?

- A Two H1s are invalid HTML
- B A page should have exactly one H1 naming its single main topic; multiple H1s make competing topic claims
- C It doubles the page weight
- D Screen readers ignore all H1s

5.1.Q7 Structure EASY An **<img>** has **alt=""** (empty). Is that always a mistake?

- A Yes, every image needs descriptive alt text
- B No — an intentionally empty alt marks a decorative image so screen readers skip it; only a missing alt attribute is a fault
- C Yes, empty alt breaks indexing
- D It depends on the file format

5.1.Q8 Transfer MEDIUM A **<meta name="robots" content="noindex, follow">** is on an important page. What is the consequence?

5.1.Q9 Recall MEDIUM Why prefer **`html.parser.HTMLParser`** over a regex for reading a **`<title>`** containing **`&`**?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 5 – On-Page Signals and the HTML Audit

5.1.Q1. `headings: 4 | mentioning keyword: 2` then `coverage: 50.0 %`

5.1.Q2. `canonical present: https://example.com/blog/post`

5.1.Q3. `signals: {'informational': 2, 'navigational': 0, 'transactional': 0}` then `inferred intent: informational`

5.1.Q4. `5 words / 2 images = 2.5 words per image -> thin (image-heavy)`

5.1.Q5. B roughly 50-60 characters keeps the title from being clipped in the results snippet (and pixel width is the truer limit).

5.1.Q6. B one H1 states the page's main topic; several blur which topic the page is really about.

5.1.Q7. B empty alt is the correct signal for decorative images; the real fault is an image with no alt attribute at all.

5.1.Q8. The `noindex` directive removes the page from the search index entirely — it will not appear in results no matter how good its content. (`follow` only governs whether its links pass credit.)

5.1.Q9. The parser decodes HTML entities (so `&` becomes &) and handles nested/malformed markup, where a naive regex leaves the entity raw and can mis-match nested tags.

Chapter 6 · Keyword Research and Intent as Data

Score, prioritise and cluster keywords; classify SERP intent; model difficulty against volume.

Chapter Skill Map

What this chapter adds: In Keyword Research and Intent as Data, the focus is score, prioritise and cluster keywords; classify serp intent; model difficulty against volume. You practise it across 35 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 6 assumes the reading habits from Chapter 5: On-Page Signals and the HTML Audit. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 7, Canonicalization, Duplicate and Thin Content, reuses this chapter's discipline with a new kind of problem.

1 **Projects 191-201**	2 **Projects 202-212**	3 **Projects 213-225**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

A keyword is not a word — it is a row of data: a query, an intent, a volume, a difficulty.

stop guessing, start scoring

Keyword research feels like brainstorming, but the moment you have a list it becomes a spreadsheet, and a spreadsheet is just rows you can filter, score, and sort. Every keyword carries the same fields: the *phrase* a user types, the *intent* behind it, an estimate of monthly search *volume*, and a *difficulty* score for how hard the first page is to win. Once those four numbers sit in a row, "which keyword should I target?" stops being an opinion and becomes a calculation.

The single most important field is not volume — it is intent. Intent is *why* someone searched. Four classes cover almost everything: informational (they want to learn — "how to clean suede shoes"), navigational (they want a specific site — "nike store login"), commercial (they are comparing before buying — "best running shoes"), and transactional (they are ready to act — "buy nike air max"). Match the wrong content to the intent and you can rank #1 and still convert no one.

The beautiful part: you can often read intent straight off the words, with no SERP and no tool. Certain trigger words give it away. "How", "what", "why", "guide" signal informational. "Buy", "price", "cheap", "order" signal transactional. "Best", "top", "review", "vs" signal commercial. A handful of `if` checks turns a raw phrase into a labelled row:

```
keywords = ['how to clean suede shoes', 'buy nike air max',
            'best running shoes', 'nike store login']
def classify(kw):
```

```
    w = kw.split()
    if any(t in w for t in ('how', 'what', 'why', 'guide')):
        return 'informational'
    if any(t in w for t in ('buy', 'price', 'cheap', 'order')):
        return 'transactional'
    if any(t in w for t in ('best', 'top', 'review', 'vs')):
        return 'commercial'
    return 'navigational'
for kw in keywords:
    print(f'{classify(kw):14} {kw}')
```

```
output
informational  how to clean suede shoes
transactional  buy nike air max
commercial     best running shoes
navigational   nike store login
```

That tiny function is the engine of the whole chapter. Once each keyword has an intent label and a couple of numbers, every "strategy" question — what to write, what to prioritise, which pages compete — reduces to filtering and sorting those rows. *label first, then sort*

SEARCH INTENT

The goal behind a query, usually one of four classes: **informational** (learn), **navigational** (reach a known site), **commercial** (compare before buying), **transactional** (act now). It decides what kind of page can rank.

SEARCH VOLUME

An estimate of how many times a keyword is searched per month. Demand, not difficulty — a big number is worthless if you can never rank for it.

KEYWORD DIFFICULTY

A 0–100 estimate of how hard it is to rank on page one, based on the strength of the pages already there. Higher means more entrenched competition.

PRIORITY SCORE

A single number that combines volume, intent value, and difficulty so a keyword list can be ranked. A common form is `volume × intent_weight ÷ difficulty`.

KEYWORD CLUSTER

A group of keywords that share intent and topic and so belong on one page. Targeting them with separate pages wastes effort and splits authority.

LONG-TAIL KEYWORD

A longer, more specific query (often 4+ words). Lower volume but lower difficulty and higher intent, so it usually converts better.

KEYWORD CANNIBALIZATION

When two or more pages on your own site target the same keyword, competing with each other and splitting rankings instead of one page winning.

Intent is a label you can read off the words

Before volume, before difficulty, decide *why* the searcher is searching, because intent decides what content can rank at all. A page that sells shoes will never win "how to clean suede shoes" — that SERP is full of guides. Intent comes in four flavours, and you can usually guess it from trigger words alone. The same trigger-word idea also maps a keyword to a funnel stage — awareness, consideration, decision — which tells you whether to write a guide, a comparison, or a pricing page:

```
def stage(kw):
    w = set(kw.split())
    if w & {'what', 'how', 'guide', 'is'}: return 'awareness'
    if w & {'best', 'top', 'review', 'vs'}: return 'consideration'
    if w & {'buy', 'price', 'plan', 'demo'}: return 'decision'
    return 'unknown'
for kw in ['what is seo', 'best seo tools', 'buy seo plan']:
    print(f'{stage(kw):14} {kw}')
```

```
output
awareness      what is seo
consideration  best seo tools
decision       buy seo plan
```

Set intersection (`w & {...}`) is the trick: split the phrase into a set of words, intersect with a set of trigger words, and a non-empty result means a match. The same pattern scales to summarising a whole list — count keywords per intent and you get a portfolio check. If a "buy" brand has 70% informational keywords, the content mix is pointed at the wrong stage of the funnel.

Volume versus difficulty: the two-axis trade-off

Every keyword lives at a point on two axes: how much traffic it could send (volume) and how hard it is to win (difficulty). High volume with high difficulty is a trap — "seo" gets 40,000 searches a month and you will never crack page one. A raw 0–100 difficulty is hard to act on, so the first move is to `band` it into easy / medium / hard, turning a number into a decision about where to start:

```
keywords = {'seo': 12, 'best seo tool': 45,
            'enterprise seo platform': 78, 'cheap seo': 30}
def band(d):
    if d < 30: return 'easy'
    if d < 60: return 'medium'
    return 'hard'
for kw in sorted(keywords):
    print(f'{band(keywords[kw]):7} ({keywords[kw]:>2})  {kw}')
```

```
output
medium  (45)  best seo tool
medium  (30)  cheap seo
hard    (78)  enterprise seo platform
easy    (12)  seo
```

The smartest one-line filter combines both axes: rank by a `volume-to-difficulty ratio` for bang-for-buck, but also impose a hard difficulty ceiling so you never chase impossible terms. The ratio alone would rank "seo" first (40000 ÷ 90 = 444), but the ceiling vetoes it — high ratio, still unreachable.

The long tail: where most queries actually live

Search demand is not concentrated in a few fat head terms. It is spread across a vast number of rare, specific phrases — the `long tail`. "Shoes" is a head term: huge volume, brutal difficulty. "Best running shoes for flat feet" is long-tail: small volume, low difficulty, and a searcher who knows exactly what they want. The crudest long-tail detector is just word count:

```
keywords = ['shoes', 'running shoes',
            'best running shoes for flat feet',
            'mens trail running shoes 2026']
for kw in keywords:
    n = len(kw.split())
    tag = 'long-tail' if n >= 4 else 'head'
    print(f'{n} words  {tag:10} {kw}')
```

```
output
1 words  head       shoes
2 words  head       running shoes
6 words  long-tail  best running shoes for flat feet
5 words  long-tail  mens trail running shoes 2026
```

A finer split adds a middle "body" band (head = 1 word, body = 2–3, long-tail = 4+). The strategic point: a hundred long-tail wins can out-traffic one impossible head term, and they convert better because the intent is sharper.

many small, specific wins beat one giant moonshot

Clustering: keywords that belong on one page

A list of 500 keywords is not 500 pages — many of them are the same topic in different words. Grouping them into clusters tells you how many pages you actually need and which keywords share a home. A fast first pass is to group by a shared token (here the last word, the head noun):

```
from collections import defaultdict
keywords = ['running shoes', 'running shorts', 'trail shoes',
            'shoes for kids', 'best shorts']
clusters = defaultdict(list)
for kw in keywords:
    clusters[kw.split()[-1]].append(kw)
for stem in sorted(clusters):
    print(f'{stem}: {sorted(clusters[stem])}')
```

```
output
kids: ['shoes for kids']
shoes: ['running shoes', 'trail shoes']
shorts: ['best shorts', 'running shorts']
```

Each cluster then earns one pillar page targeting its strongest keyword. The deeper test of whether two keywords share a cluster is whether their SERPs look alike: if "seo tools" and "best seo tools" return mostly the same ranking domains, they share intent and should be one page, not two.

Cannibalization: when your own pages compete

The opposite failure of clustering is keyword cannibalization — two of your *own* pages chasing the same keyword. Google does not know which to rank, so it splits signals between them and neither wins. You catch it by inverting your page→keyword map into keyword→pages and flagging any keyword owned by more than one URL:

```
from collections import defaultdict
pages = {'/running-shoes': 'best running shoes',
         '/shoes/running': 'best running shoes',
         '/trail-shoes': 'trail running shoes'}
target = defaultdict(list)
for url, kw in pages.items():
    target[kw].append(url)
for kw in sorted(target):
    urls = sorted(target[kw])
    if len(urls) > 1:
        print(f'CANNIBALIZATION: {kw!r} -> {urls}')
```

```
    else:
        print(f'ok: {kw!r} -> {urls[0]}')
```

```
output
CANNIBALIZATION: 'best running shoes' -> ['/running-shoes', '/shoes/running']
ok: 'trail running shoes' -> /trail-shoes
```

The fix is to merge or redirect: pick one canonical page per keyword and consolidate. The whole chapter now composes — classify intent, weight volume by it, divide by difficulty — into a single ranked decision table, the heart of a keyword strategy:

```
weights = {'informational': 0.5, 'commercial': 1.5,
           'transactional': 2.0, 'navigational': 0.3}
def intent(kw):
    w = set(kw.split())
    if w & {'buy', 'price', 'order'}: return 'transactional'
    if w & {'best', 'top', 'review'}: return 'commercial'
    if w & {'what', 'how', 'why'}: return 'informational'
    return 'navigational'
rows = [('best running shoes', 8000, 45),
        ('buy running shoes online', 3000, 30),
        ('nike store', 12000, 70)]
scored = []
for kw, vol, diff in rows:
    i = intent(kw)
    scored.append((round(vol * weights[i] / diff, 1), i, kw))
for score, i, kw in sorted(scored, reverse=True):
    print(f'{score:>7}  {i:14} {kw}')
```

```
output
  266.7  commercial     best running shoes
  200.0  transactional  buy running shoes online
   51.4  navigational   nike store
```

"nike store" has the most volume (12,000) yet ranks last: it is navigational (people want Nike's own site, weight 0.3) and difficult (70). The score sees what raw volume cannot.

volume is the loudest number and the most misleading

Example 1

Score and rank three keywords with priority = volume × intent_weight ÷ difficulty. "running shoes" (vol 9000, diff 60, weight 2); "best running shoes 2026" (vol 4000, diff 30, weight 3); "what are running shoes" (vol 1200, diff 15, weight 1).

SOLUTION

1. "running shoes": 9000 × 2 ÷ 60 = 18000 ÷ 60 = `300.0`.
2. "best running shoes 2026": 4000 × 3 ÷ 30 = 12000 ÷ 30 = `400.0`.
3. "what are running shoes": 1200 × 1 ÷ 15 = 1200 ÷ 15 = `80.0`.
4. Sort descending by score: 400.0, then 300.0, then 80.0.

Ranked: best running shoes 2026 (400.0) > running shoes (300.0) > what are running shoes (80.0). The high-intent, low-difficulty term beats the higher-volume one. *biggest volume did not win*

Example 2

Two of your pages both target "best running shoes": /running-shoes and /shoes/running. A third page /trail-shoes targets "trail running shoes". Which keyword is cannibalized, and what is the fix?

SOLUTION

1. Invert page→keyword into keyword→pages. "best running shoes" maps to `both /running-shoes and /shoes/running`.
2. "trail running shoes" maps to a single page, /trail-shoes — no conflict.
3. A keyword owned by more than one URL is cannibalized: your pages compete and split rankings.

"best running shoes" is cannibalized (two pages target it). Fix: pick one canonical page and merge or 301-redirect the other into it. *one keyword, one home*

Example 3

Classify the intent of "cheap trainers price" and "how to lace running shoes" using trigger words.

SOLUTION

1. "cheap trainers price" contains `cheap` and `price` — both transactional triggers (ready to buy).
2. "how to lace running shoes" starts with `how` — an informational trigger (wants to learn).
3. Match content to each: a product/category page for the first, a how-to guide for the second.

Transactional and informational respectively. Same product space, opposite content — sell to one, teach the other. *intent, not topic, picks the page type*

Mental-Math Shortcut

One keyword, one row, one home. Read the intent off the words first, then turn the rest into numbers — `volume × intent_weight ÷ difficulty` — and *sort the rows*. The highest-volume keyword is the loudest and the most misleading; let the score, not the big number, pick your targets. And every keyword you keep gets exactly one page that owns it, or your own pages cannibalize each other.

Common Mistakes

- Chasing volume and ignoring intent — ranking #1 for an informational query with a product page converts no one, because the searcher came to learn, not to buy.
- Picking keywords by volume alone, so you target high-volume head terms whose difficulty means you never reach page one.
- Building a separate page for every keyword instead of clustering — you fragment authority and end up with thin, overlapping pages competing with each other.
- Letting two pages target the same keyword (cannibalization) and wondering why neither ranks; the fix is to merge or redirect to one canonical page.

This chapter contains 35 hands-on mini-projects (Projects 191–225). Type each one out, run it, and experiment.

Project 191 Score and Rank Keywords by Priority

The Idea Behind It

This project turns three keyword dicts into a comparable shortlist. A loop computes score as volume times intent_weight divided by difficulty, rounded to one decimal, then sorted with a negative-score key and keyword tiebreak ranks them. The formula deliberately rewards high demand and strong intent while penalising contested terms, so one number captures the trade-off you weigh by hand.

What You'll Build & Learn

Collapse volume, intent, and difficulty into a single priority score, then print keywords ranked best-first.

- Storing keyword metrics in dictionaries
- A weighted scoring formula in code
- Sorting with a negative-key descending order
- Stable tiebreaks using a secondary key

Skill Roadmap

Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 192 changes the surface problem while preserving the underlying method. *progress check*

Reading target: score and rank keywords by priority is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f"{k['score']:>6} {k['keyword']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for k in keywords:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{k['score']:>6} {k['keyword']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for k in keywords:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score and Rank Keywords by Priority: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, running, shoes, volume. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = [| Core: for k in keywords: | Evidence: print(f"{k['score']:>6} {k['keyword']}")

🛠️ Reader to Writer Bridge

1. Read: explain score and rank keywords by priority in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 192 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{k['score']:>6} {k['keyword']}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of score and rank keywords by priority and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Each keyword gets a derived `score` field, then sorting by negative score (with the keyword as a deterministic tiebreaker) puts the highest-value opportunity first.

```
keywords = [
    {'keyword': 'running shoes', 'volume': 9000, 'difficulty': 60,
'intent_weight': 2},
    {'keyword': 'best running shoes 2026', 'volume': 4000, 'difficulty': 30,
'intent_weight': 3},
    {'keyword': 'what are running shoes', 'volume': 1200, 'difficulty': 15,
'intent_weight': 1},
]
for k in keywords:
    k['score'] = round(k['volume'] * k['intent_weight'] / k['difficulty'], 1)
ranked = sorted(keywords, key=lambda k: (-k['score'], k['keyword']))
for k in ranked:
    print(f"{k['score']:>6}  {k['keyword']}")
```

```
output
 400.0  best running shoes 2026
 300.0  running shoes
  80.0  what are running shoes
```

How it works:

1. A list of dicts holds keyword, volume, difficulty, and intent_weight for three running-shoe terms.
2. The for loop writes a new score key into each dict using the volume times intent_weight over difficulty formula.
3. sorted uses a lambda returning a tuple of negative score then keyword, so higher scores rank first with alphabetical tiebreaks.
4. A final loop prints the right-aligned score beside each keyword in priority order.

Try it:

1. Add an estimated cost-per-click field and fold it into the score formula.
2. Print only keywords scoring above a chosen threshold instead of all of them.
3. Swap the difficulty divisor for the square root of difficulty to soften its effect.
4. Add a keyword with volume 20000 and difficulty 90 and see where it lands.

⚠ Common Mistakes

- Dividing by difficulty without guarding against a zero difficulty would raise ZeroDivisionError.
- Sorting by `k['score']` ascending lists the worst keyword first.
- Dropping the `k['keyword']` tiebreaker makes two equal scores order unpredictably.

Skills gained dict scoring | sorted with lambda | f-string alignment

→ Where You'll Use This: Every keyword research deliverable ends in a ranked shortlist; this scoring pattern is the spine of any prioritisation spreadsheet handed to a content team.

Mental Model

Like a recipe that boils three ingredients down to one flavour score, so you taste-test keywords against each other instead of guessing.

Project 192 Classify Search Intent by Trigger Words

The Idea Behind It

Here a classify function reads the words of each keyword and returns an intent label by checking membership against ordered trigger-word tuples. Question words mean informational, purchase words mean transactional, comparison words mean commercial, and anything left falls through to navigational. Splitting first means whole-word matching, so it never misfires on a substring buried inside another term.

What You'll Build & Learn

Label each keyword with a search intent using only trigger words, no SERP lookup required.

- Search intent from trigger vocabulary
- any() over a tuple of triggers
- Order of checks acts as priority
- A default category for leftovers

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 191: score and rank keywords by priority. Project 193 changes the surface problem while preserving the underlying method.

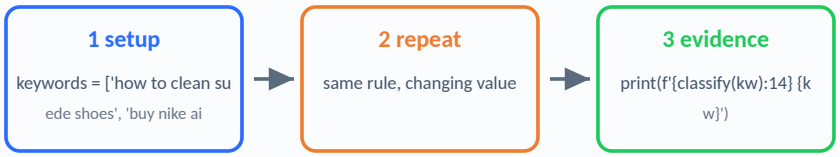
progress check

Reading target: classify search intent by trigger words is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = ['how to clean suede shoes', 'buy nike air max', 'best running shoes', 'nike store login', 'cheap trainers price']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{classify(kw):14} {kw}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = ['how to clean suede shoes', 'buy nike air max', 'best running shoes', 'nike store login', 'cheap trainers price']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if any(t in w for t in ('how', 'what', 'why', 'guide', 'tutorial')):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{classify(kw):14} {kw}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if any(t in w for t in ('how', 'what', 'why', 'guide', 'tutorial')):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Classify Search Intent by Trigger Words: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, how, to, clean, suede. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = ['how to clean suede shoes', 'buy nike air max', 'best runni... | Core: if any(t in w for t in ('how', 'what', 'why', 'guide', 'tutorial')): | Evidence: print(f'{classify(kw):14} {kw}')

🛠️ Reader to Writer Bridge

1. Read: explain classify search intent by trigger words in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 191.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 193 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{classify(kw):14} {kw}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = ['how to clean suede shoes', 'buy nike air max', 'best running shoes', 'nike store login', 'cheap trainers price']` and say

why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of classify search intent by trigger words and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Checking each keyword's words against ordered trigger sets assigns the first matching intent, with navigational as the catch-all default.

```
keywords = ['how to clean suede shoes', 'buy nike air max', 'best running
shoes', 'nike store login', 'cheap trainers price']
def classify(kw):
    w = kw.split()
    if any(t in w for t in ('how', 'what', 'why', 'guide', 'tutorial')):
        return 'informational'
    if any(t in w for t in ('buy', 'price', 'cheap', 'order', 'discount')):
        return 'transactional'
    if any(t in w for t in ('best', 'top', 'review', 'vs', 'compare')):
        return 'commercial'
    return 'navigational'
for kw in keywords:
    print(f'{classify(kw):14} {kw}')
```

```
output
informational  how to clean suede shoes
transactional  buy nike air max
commercial     best running shoes
navigational   nike store login
transactional  cheap trainers price
```

How it works:

1. A list of five mixed keywords feeds the classify function.
2. classify splits the keyword into words, then uses any() to test if any trigger from each tuple appears in that word list.
3. The checks run informational, transactional, then commercial; the first match wins, and unmatched keywords return navigational.
4. The output loop prints each left-padded label beside its keyword.

Try it:

1. Add a transactional trigger for 'coupon' and 'deal' and retest the list.
2. Return all matching intents as a list instead of stopping at the first.
3. Make the navigational fallback only fire when a known brand word is present.

4. Add 'near me' handling to flag local intent separately.

⚠ Common Mistakes

- Order matters: checking 'best' before 'buy' would misclassify 'buy the best phone' as commercial.
- Using `t in kw` (substring) instead of `t in w` (whole word) matches 'how' inside 'show'.
- Forgetting the default return leaves keywords with no trigger word unclassified (None).

Skills gained intent classification | tuple membership | early return

→ Where You'll Use This: Bulk intent tagging is the first column SEOs add to a raw keyword export, deciding whether each term needs a guide, a comparison, or a product page.

Mental Model

Like a postal sorter glancing at one word on each envelope to drop it into the right bin without reading the whole letter.

Project 193 Cluster Keywords by Shared Head Term

The Idea Behind It

Grouping keywords by their final word is the trick here. A defaultdict of lists takes each keyword, splits it, grabs the last token as a crude head term, and appends the phrase under that stem. Sorting the stems then printing each sorted group reveals topic clusters forming around shared nouns. It is blunt but instantly shows which phrases want to live together.

What You'll Build & Learn

Bucket keywords into emerging topic clusters keyed on their last word.

- defaultdict(list) for grouping
- Using the last token as a stem
- Sorting keys and values for stable output

Skill Roadmap

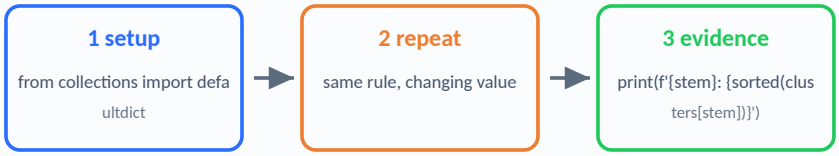
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 192: classify search intent by trigger words. Project 194 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cluster keywords by shared head term is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{stem}: {sorted(clusters[stem])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`keywords = ['running shoes', 'running shorts', 'trail shoes', 'shoes for kids', 'best shorts']`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{stem}: {sorted(clusters[stem])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
keywords = ['running shoes', 'running shorts', 'trail shoes', 'shoes for kids', 'best shorts']
```

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Cluster Keywords by Shared Head Term: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: collections, defaultdict, keywords, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: keywords = ['running shoes', 'running shorts', 'trail shoes', 'shoes fo... | Evidence: print(f'{stem}: {sorted(clusters[stem])}')

🛠️ Reader to Writer Bridge

1. Read: explain cluster keywords by shared head term in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 192.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 194 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{stem}: {sorted(clusters[stem])}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of cluster keywords by shared head term and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A `defaultdict(list)` keyed on the final token groups keywords sharing a head noun; sorting the keys and members keeps the output stable.

```
from collections import defaultdict
keywords = ['running shoes', 'running shorts', 'trail shoes', 'shoes for kids',
'best shorts']
clusters = defaultdict(list)
for kw in keywords:
    stem = kw.split()[-1]
    clusters[stem].append(kw)
for stem in sorted(clusters):
    print(f'{stem}: {sorted(clusters[stem])}')
```

```
output
kids: ['shoes for kids']
shoes: ['running shoes', 'trail shoes']
shorts: ['best shorts', 'running shorts']
```

How it works:

1. A defaultdict(list) named clusters auto-creates an empty list for any new stem.
2. The loop splits each keyword and takes index [-1], the last word, as the grouping stem.
3. Each keyword is appended to the list under its stem.
4. Iterating sorted stems prints each cluster with its members sorted alphabetically.

Try it:

1. Cluster on the first word instead of the last and compare the groupings.
2. Skip keywords whose stem is a stopword like 'the' or 'for'.
3. Print only clusters that contain two or more keywords.
4. Cluster on the first word instead and compare the groupings.

⚠ Common Mistakes

- Using the last word naively misgroups 'shoes for kids' under 'kids' instead of 'shoes'.
- A plain dict would raise KeyError on first insert without `setdefault` or `defaultdict`.

- Forgetting to sort the cluster members leaves their order at the mercy of input order.

Skills gained keyword clustering | defaultdict | string split

→ Where You'll Use This: Quick last-word clustering gives a first-pass map of which keywords belong on the same page before you invest in heavier semantic clustering tools.

Mental Model

Like sorting socks by colour: a rough rule that gets matching pairs near each other fast, even if a few oddballs end up in the wrong pile.

Project 194 Bucket Difficulty into Easy, Medium, Hard

The Idea Behind It

A band function maps a numeric difficulty into easy, medium, or hard using two thresholds at 30 and 60. The keywords dict pairs terms with their scores; iterating the sorted keys looks up each difficulty and prints the band, the padded number, and the keyword. Turning a continuous 0-to-100 metric into three labels makes the data instantly actionable instead of just numeric.

What You'll Build & Learn

Convert raw difficulty scores into easy, medium, and hard labels for fast triage.

- Threshold banding of a numeric metric
- Cascading if checks as boundaries
- Looking up values by sorted key

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 193: cluster keywords by shared head term. Project 195 changes the surface problem while preserving the underlying method. *progress check*

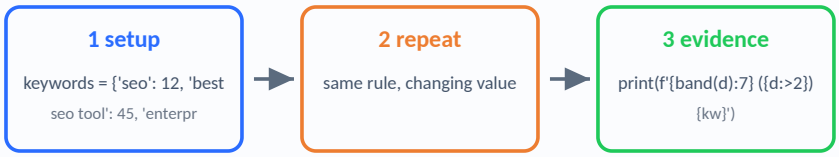
Reading target: bucket difficulty into easy, medium, hard is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = {'seo': 12, 'best seo tool': 45, 'enterprise seo platform': 78, 'cheap seo': 30}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{band(d):7} ({d:>2}) {kw}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = {'seo': 12, 'best seo tool': 45, 'enterprise seo platform': 78, 'cheap seo': 30}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if d < 30: return 'easy'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{band(d):7} ({d:>2}) {kw}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if d < 30: return 'easy'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Bucket Difficulty into Easy, Medium, Hard: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, seo, best, tool, enterprise. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = {'seo': 12, 'best seo tool': 45, 'enterprise seo platform':... | Core: if d < 30: return 'easy' | Evidence: print(f'{band(d):7} ({d:>2}) {kw}')

🛠️ Reader to Writer Bridge

1. Read: explain bucket difficulty into easy, medium, hard in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 193.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 195 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{band(d):7} ({d:>2}) {kw}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = {'seo': 12, 'best seo tool': 45, 'enterprise seo platform': 78, 'cheap seo': 30}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of bucket difficulty into easy, medium, hard and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Cascading `if` thresholds map a continuous difficulty into three named bands; the first matching range wins so order of the cutoffs matters.

```
keywords = {'seo': 12, 'best seo tool': 45, 'enterprise seo platform': 78,
'cheap seo': 30}
def band(d):
    if d < 30: return 'easy'
    if d < 60: return 'medium'
    return 'hard'
for kw in sorted(keywords):
    d = keywords[kw]
    print(f'{band(d):7} ({d:>2})  {kw}')
```

```
output
medium  (45)  best seo tool
medium  (30)  cheap seo
hard    (78)  enterprise seo platform
easy    (12)  seo
```

How it works:

1. The keywords dict maps four SEO phrases to integer difficulty scores.
2. band returns 'easy' below 30, 'medium' below 60, otherwise 'hard', using ordered if statements as cut points.
3. The loop walks the dict's sorted keys, reads each difficulty, and calls band on it.
4. Output prints the padded band, the difficulty in parentheses, and the keyword.

Try it:

1. Add a fourth 'very hard' band for scores at or above 80.
2. Make the thresholds function arguments so they can be tuned per client.
3. Sort the output by difficulty instead of alphabetically by keyword.
4. Add a fourth band 'very hard' for difficulty 80 and above.

⚠ Common Mistakes

- A difficulty of exactly 30 lands in 'medium' because the test is `< 30`, not `<= 30`.
- Reordering the `if` checks (hard first) would mislabel low scores.
- Iterating the dict without sorting gives nondeterministic output order.

Skills gained difficulty banding | threshold logic | dict iteration

→ Where You'll Use This: Difficulty bands appear in nearly every keyword report so non-technical stakeholders can see at a glance where the quick wins versus the long campaigns live.

Mental Model

Like a ski slope's green, blue, and black diamond markers: one symbol tells you whether to attempt the run without reading a gradient chart.

Project 195 Detect Long-Tail Keywords by Word Count

The Idea Behind It

This counts the words in each keyword and tags it long-tail when four or more words are present, otherwise head. The split-and-len pattern is the whole detector: more words means a more specific, lower-competition query. Printing the word count next to the tag and the phrase shows directly why a six-word term reads as long-tail while a two-word phrase stays a head term.

What You'll Build & Learn

Flag long-tail keywords by counting words, separating specific phrases from broad head terms.

- Word count as a length proxy
- Inline conditional expression for tagging
- split() to tokenise a phrase

Skill Roadmap

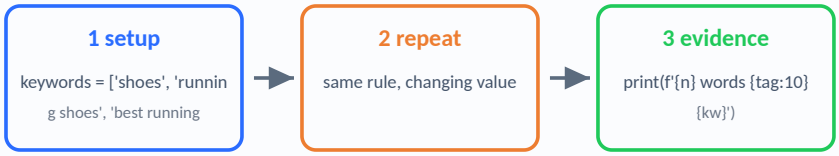
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 194: bucket difficulty into easy, medium, hard. Project 196 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect long-tail keywords by word count is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = ['shoes', 'running shoes', 'best running shoes for flat feet', 'mens trail running shoes 2026']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{n} words {tag:10} {kw}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = ['shoes', 'running shoes', 'best running shoes for flat feet', 'mens trail running shoes 2026']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`tag = 'long-tail' if n >= 4 else 'head'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{n} words {tag:10} {kw}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `tag = 'long-tail' if n >= 4 else 'head'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Long-Tail Keywords by Word Count: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, shoes, running, best, flat. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = ['shoes', 'running shoes', 'best running shoes for flat feet... | Core: tag = 'long-tail' if n >= 4 else 'head' | Evidence: print(f'{n} words {tag:10} {kw}')

🛠️ Reader to Writer Bridge

1. Read: explain detect long-tail keywords by word count in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 194.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 196 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{n} words {tag:10} {kw}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = ['shoes', 'running shoes', 'best running shoes for flat feet', 'mens trail running shoes 2026']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect long-tail keywords by word count and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Splitting on whitespace and counting tokens separates short head terms from descriptive long-tail phrases at a four-word threshold.

```
keywords = ['shoes', 'running shoes', 'best running shoes for flat feet', 'mens
trail running shoes 2026']
for kw in keywords:
    n = len(kw.split())
```

```
    tag = 'long-tail' if n >= 4 else 'head'
    print(f'{n} words  {tag:10} {kw}')
```

```
output
1 words  head       shoes
2 words  head       running shoes
6 words  long-tail  best running shoes for flat feet
5 words  long-tail  mens trail running shoes 2026
```

How it works:

1. A list holds four keywords of varying length.
2. For each, len(kw.split()) counts the words into n.
3. An inline conditional sets tag to 'long-tail' when n is four or more, else 'head'.
4. The print shows the count, the padded tag, and the keyword.

Try it:

1. Lower the long-tail threshold to three words and see what flips.
2. Add a third 'mid-tail' tag for exactly two or three words.
3. Count characters instead of words and compare the classifications.
4. Lower the threshold to 3 and see which keywords flip.

⚠ Common Mistakes

- Counting characters instead of words flags any long single word as long-tail.
- A keyword with double spaces would over-count words unless you normalise whitespace.
- Hard-coding the threshold at 3 vs 4 quietly reclassifies three-word terms.

Skills gained long-tail detection | word counting | ternary expression

→ Where You'll Use This: Long-tail flagging steers content planning toward specific, easier-to-rank, higher-converting queries that often beat chasing crowded head terms.

Mental Model

Like judging a question's specificity by its length: 'shoes?' is vague, but 'best trail shoes for flat feet?' tells you exactly what they want.

Project 196 Detect Keyword Cannibalization

The Idea Behind It

Cannibalization happens when several URLs chase the same keyword and split each other's rankings. The code inverts a page-to-keyword dict into a defaultdict mapping each target keyword to the list of URLs aiming at it.

Any keyword with more than one URL is printed with a CANNIBALIZATION warning; single-owner keywords print as ok. The inversion is what surfaces the hidden conflict.

What You'll Build & Learn

Detect keyword cannibalization by finding any keyword targeted by more than one page.

- Inverting a dict to group by value
- defaultdict(list) for many-to-one mapping
- Branching on list length to flag conflicts

🧭 Skill Roadmap

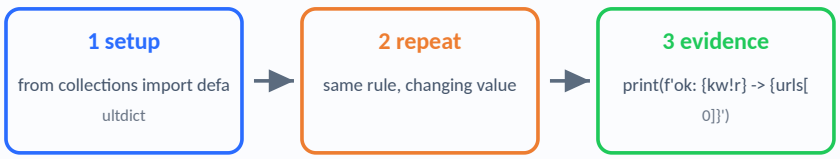
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 195: detect long-tail keywords by word count. Project 197 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect keyword cannibalization is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'ok: {kw!r} -> {urls[0]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if len(urls) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'ok: {kw!r} -> {urls[0]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(urls) > 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Keyword Cannibalization: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: collections, defaultdict, pages, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: if len(urls) > 1: | Evidence: print(f'ok: {kw!r} -> {urls[0]}')

🛠️ Reader to Writer Bridge

1. Read: explain detect keyword cannibalization in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 195.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 197 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'ok: {kw!r} -> {urls[0]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect keyword cannibalization and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Inverting the page-to-keyword map into keyword-to-URLs reveals any keyword claimed by more than one page, which is the definition of cannibalization.

```
from collections import defaultdict
pages = {
    '/running-shoes': 'best running shoes',
    '/shoes/running': 'best running shoes',
    '/trail-shoes': 'trail running shoes',
}
target = defaultdict(list)
for url, kw in pages.items():
    target[kw].append(url)
for kw in sorted(target):
    urls = sorted(target[kw])
    if len(urls) > 1:
        print(f'CANNIBALIZATION: {kw!r} targeted by {urls}')
    else:
        print(f'ok: {kw!r} -> {urls[0]}')
```

```
output
CANNIBALIZATION: 'best running shoes' targeted by ['/running-shoes',
'/shoes/running']
ok: 'trail running shoes' -> /trail-shoes
```

How it works:

1. The pages dict maps three URLs to the keyword each one targets.
2. A defaultdict(list) called target collects every URL under its keyword as the loop walks pages.items().
3. Sorting keywords, the code checks if a keyword's URL list has more than one entry.
4. Multi-URL keywords print a CANNIBALIZATION line; single-URL keywords print an ok line.

Try it:

1. Report the count of conflicting URLs alongside each warning.
2. Read the page-to-keyword pairs from a CSV file instead of a literal dict.
3. Suggest keeping the shortest URL and redirecting the rest.
4. Add a third page also targeting 'best running shoes' and watch the conflict grow.

⚠ Common Mistakes

- Building the map keyword-first instead of inverting it misses that pages, not keywords, are the keys.
- Comparing raw keyword strings without normalising case treats 'Shoes' and 'shoes' as different targets.
- Reporting a single-URL keyword as a conflict by checking `>= 1` instead of `> 1`.

Skills gained cannibalization audit | dict inversion | conditional reporting

→ Where You'll Use This: Cannibalization checks run during site audits to find pages competing against each other so you can consolidate, redirect, or re-target the weaker ones.

⚡ Mental Model

Like two sales reps cold-calling the same client: instead of one strong pitch, the prospect gets two diluted ones and trust splits between them.

Project 197 Compute Term Frequency for a Seed Across Pages

💡 The Idea Behind It

Term frequency measures how densely a page mentions a seed word, normalised by length so long pages are not unfairly favoured. The code splits each document body, counts how often the term appears, and divides by total word count. Page p1, packed with 'seo', scores 0.6, while p3 that never mentions it scores 0.0, giving a clean on-topic signal per page.

What You'll Build & Learn

Compute normalised term frequency of a seed keyword across several page bodies.

- Term frequency as count over length
- list.count() for token tallying

- Why normalisation beats raw counts

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 196: detect keyword cannibalization. Project 198 changes the surface problem while preserving the underlying method.

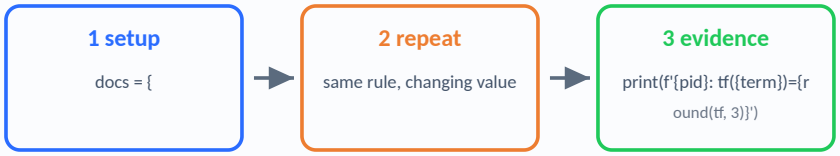
progress check

Reading target: compute term frequency for a seed across pages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `docs = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{pid}: tf({term})={round(tf, 3)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`docs = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for pid in sorted(docs):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'{pid}: tf({term})={round(tf, 3)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for pid in sorted(docs):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute Term Frequency for a Seed Across Pages: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: docs, p1, seo, audit, report. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: docs = { | Core: for pid in sorted(docs): | Evidence: print(f'{pid}: tf({term})={round(tf, 3)}')

Reader to Writer Bridge

1. Read: explain compute term frequency for a seed across pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 196.
   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 198 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{pid}: tf({term})={round(tf, 3)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `docs = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute term frequency for a seed across pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Dividing the seed's occurrence count by the document's token count gives term frequency, a length-normalised density that beats a raw count.

```
docs = {
    'p1': 'seo audit seo report seo',
    'p2': 'content marketing seo guide',
    'p3': 'link building tactics',
}
term = 'seo'
for pid in sorted(docs):
    body = docs[pid].split()
    tf = body.count(term) / len(body)
    print(f'{pid}: tf({term})={round(tf, 3)}')
```

```
output
p1: tf(seo)=0.6
p2: tf(seo)=0.25
p3: tf(seo)=0.0
```

How it works:

1. A docs dict maps three page ids to their text bodies, and term is set to 'seo'.
2. For each sorted page id, the body is split into a word list.
3. tf is the body's count of term divided by the body's length.
4. Each page id prints with its rounded term frequency.

Try it:

1. Lowercase each body before counting so 'SEO' and 'seo' both count.
2. Compute term frequency for a list of seed terms at once.

3. Flag any page whose term frequency exceeds a stuffing threshold.
4. Switch the seed to 'building' and re-run.

⚠ Common Mistakes

- Using a raw count instead of dividing by length favours longer documents.
- Counting before lowercasing makes 'SEO' and 'seo' count as different terms.
- Dividing by zero on an empty document body would raise an error.

Skills gained term frequency | text normalisation | string counting

→ Where You'll Use This: Term frequency underpins on-page content scoring, helping you judge whether a page covers its target keyword densely enough without over-stuffing it.

Mental Model

Like measuring the salt concentration of soup rather than total grains: a big pot and a small cup can taste equally salty once you account for volume.

Project 198 Measure Keyword-Set Overlap with Jaccard

The Idea Behind It

Jaccard similarity compresses how much two keyword sets overlap into one number between zero and one. The code takes set intersection and union, then divides their sizes. With two of five total keywords shared, the result is 0.4. Using Python sets makes the intersection and union operators do all the heavy lifting, and printing the shared members shows exactly what drives the score.

What You'll Build & Learn

Measure overlap between two keyword sets with a single Jaccard similarity number.

- Jaccard as intersection over union
- Set operators & and | in Python
- Quantifying list overlap as a ratio

Skill Roadmap

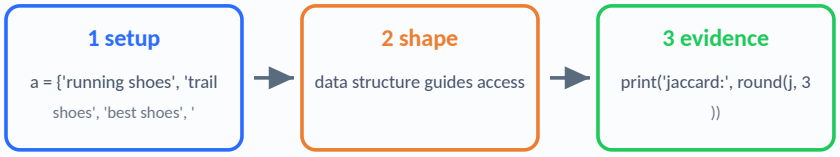
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 197: compute term frequency for a seed across pages. Project 199 changes the surface problem while preserving the underlying method. *progress check*

Reading target: measure keyword-set overlap with jaccard is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `a = {'running shoes', 'trail shoes', 'best shoes', 'cheap shoes'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The data structure is doing part of the thinking. Ask what the collection stores, how items are found, and what order or uniqueness it promises.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print('jaccard:', round(j, 3))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, shape applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`a = {'running shoes', 'trail shoes', 'best shoes', 'cheap shoes'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`j = len(inter) / len(union)`	This line depends on the chosen data shape. Ask how items are stored, found, or ordered. *slow line*
3. Check the evidence	`print('jaccard:', round(j, 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `j = len(inter) / len(union)`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Name the collection shape, then ask how an item is added, selected, transformed, or counted.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Measure Keyword-Set Overlap with Jaccard: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: running, shoes, trail, best, cheap. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: a = {'running shoes', 'trail shoes', 'best shoes', 'cheap shoes'} | Core: j = len(inter) / len(union) | Evidence: print('jaccard:', round(j, 3))

Reader to Writer Bridge

1. Read: explain measure keyword-set overlap with jaccard in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 197.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 199 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print('jaccard:', round(j, 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `a = {'running shoes', 'trail shoes', 'best shoes', 'cheap shoes'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of measure keyword-set overlap with jaccard and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Set intersection finds shared keywords and set union finds the total distinct keywords; their size ratio is the Jaccard index between 0 and 1.

```
a = {'running shoes', 'trail shoes', 'best shoes', 'cheap shoes'}
b = {'running shoes', 'best shoes', 'walking shoes'}
inter = a & b
union = a | b
j = len(inter) / len(union)
print('shared:', sorted(inter))
print('jaccard:', round(j, 3))
```

```
output
shared: ['best shoes', 'running shoes']
jaccard: 0.4
```

How it works:

1. Two sets a and b hold keyword phrases, with some shared and some unique.
2. inter is a & b, the shared keywords; union is a | b, every keyword across both.
3. j divides the size of inter by the size of union to get the Jaccard score.
4. The code prints the sorted shared keywords and the rounded similarity.

Try it:

1. Print the keywords unique to each set as a gap list.
2. Compare three sets pairwise and report all similarity scores.
3. Lowercase and strip every keyword before building the sets.
4. Add 'trail shoes' to set b and watch the index rise.

⚠ Common Mistakes

- Dividing by `len(a) + len(b)` double-counts the shared keywords in the denominator.
- Using lists instead of sets makes `&` and `|` invalid operations.
- Forgetting to sort the intersection before printing breaks deterministic output.

Skills gained Jaccard similarity | set operations | overlap ratio

→ Where You'll Use This: Set-overlap scores quantify how much your keyword coverage mirrors a competitor's, guiding gap analysis and showing where your strategy diverges.

Mental Model

Like overlapping two Venn-diagram circles and asking what fraction of the total area sits in the shared lens, the bigger the lens the more alike the lists.

Project 199 Group Keywords by Assigned Topic

The Idea Behind It

When a keyword export already carries a topic label, rolling keywords up by that label shows which themes have enough demand for a pillar page. A defaultdict of lists appends each keyword under its given topic, and the output prints each topic with its member count and sorted keywords. The grouping is driven entirely by the supplied label, not by guessing the topic.

What You'll Build & Learn

Roll keywords up by a pre-assigned topic label and report the size of each group.

- Grouping by an existing label field
- defaultdict for accumulating members
- Reporting group size alongside contents

Skill Roadmap

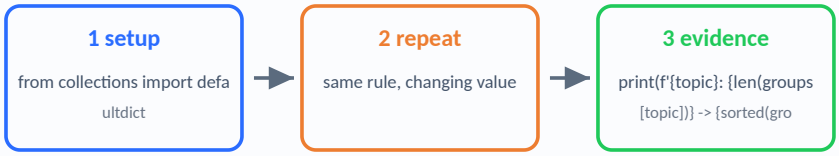
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 198: measure keyword-set overlap with jaccard. Project 200 changes the surface problem while preserving the underlying method. *progress check*

Reading target: group keywords by assigned topic is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{topic}: {len(groups[topic])} -> {sorted(groups[topic])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for kw, topic in topics.items():`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{topic}: {len(groups[topic])} -> {sorted(groups[topic])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for kw, topic in topics.items():`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Group Keywords by Assigned Topic: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: collections, defaultdict, topics, on, page. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: for kw, topic in topics.items(): | Evidence: print(f'{topic}: {len(groups[topic])} -> {sorted(groups[topic])}')

🛠️ Reader to Writer Bridge

1. Read: explain group keywords by assigned topic in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 198.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 200 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{topic}: {len(groups[topic])} -> {sorted(groups[topic])}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of group keywords by assigned topic and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Iterating the keyword-to-topic map and appending into a `defaultdict(list)` inverts it into topic-to-keywords, the basis of a content cluster plan.

```
from collections import defaultdict
topics = {
    'on page seo': 'on-page',
    'meta description length': 'on-page',
    'backlink audit': 'off-page',
```

```
    'guest posting': 'off-page',
    'core web vitals': 'technical',
}
groups = defaultdict(list)
for kw, topic in topics.items():
    groups[topic].append(kw)
for topic in sorted(groups):
    print(f'{topic}: {len(groups[topic])} -> {sorted(groups[topic])}')
```

```
output
off-page: 2 -> ['backlink audit', 'guest posting']
on-page: 2 -> ['meta description length', 'on page seo']
technical: 1 -> ['core web vitals']
```

How it works:

1. The topics dict maps five keywords to topic labels like on-page or off-page.
2. A defaultdict(list) named groups appends each keyword under its topic as the loop runs.
3. Iterating sorted topics, the code reads each group's list.
4. Each line prints the topic, its member count, and the sorted keywords.

Try it:

1. Sort topics by member count so the biggest theme prints first.
2. Flag any topic with fewer than two keywords as too thin for a pillar.
3. Sum a volume field per topic instead of just counting keywords.
4. Add a 'local' topic with two keywords and see it appear.

⚠ Common Mistakes

- Swapping key and value when iterating groups by keyword instead of topic.
- Printing `groups` directly leaks the defaultdict repr instead of clean lines.
- Not sorting either the topics or the members yields run-to-run variation.

Skills gained topic grouping | defaultdict | group counting

→ Where You'll Use This: Topic roll-ups turn a flat keyword sheet into a content plan, revealing which themes carry enough keywords to justify a dedicated pillar or hub page.

Mental Model

Like tallying votes into party buckets: once each ballot is filed under its label, the size of each pile tells you where the support concentrates.

Project 200 Score Opportunity from Volume and Current Rank

The Idea Behind It

The biggest wins are high-volume keywords where you already rank but not on page one. This opportunity score multiplies volume by rank, but only when rank is worse than ten, otherwise it scores zero because page-one terms need no rescue. 'keyword research' at rank 25 tops the list while a rank-4 term scores zero. Sorting by negative opportunity surfaces the climbable giants first.

What You'll Build & Learn

Score and rank keywords by upside, favouring high-volume terms stuck just off page one.

- Opportunity as volume times poor rank
- Conditional scoring with a rank gate
- Zeroing out already-strong positions

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 199: group keywords by assigned topic. Project 201 changes the surface problem while preserving the underlying method.

progress check

Reading target: score opportunity from volume and current rank is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

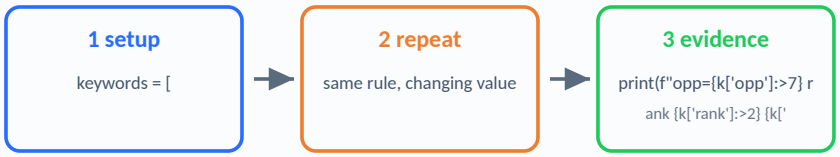
- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f"opp={k['opp']:>7} rank {k['rank']:>2} {k['keyword']}")` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`k['opp'] = round(k['volume'] * k['rank'] / 100, 1) if k['rank'] > 10 else 0`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"opp={k['opp']:>7} rank {k['rank']:>2} {k['keyword']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `k['opp'] = round(k['volume'] * k['rank'] / 100, 1) if k['rank'] > 10 else 0`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score Opportunity from Volume and Current Rank: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, seo, tools, volume. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = [| Core: k['opp'] = round(k['volume'] * k['rank'] / 100, 1) if k['rank'] > 10 el… | Evidence: print(f"opp={k['opp']:>7} rank {k['rank']:>2} {k['keyword']}")

🛠️ Reader to Writer Bridge

1. Read: explain score opportunity from volume and current rank in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 199.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 201 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"opp={k['opp']:>7} rank {k['rank']:>2} {k['keyword']}")`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score opportunity from volume and current rank and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Keywords already ranking in the top ten get zero opportunity (the win is captured), while those further down score in proportion to volume and how far they sit from page one.

```
keywords = [
    {'keyword': 'seo tools', 'volume': 8000, 'rank': 18},
    {'keyword': 'free seo audit', 'volume': 5000, 'rank': 4},
    {'keyword': 'keyword research', 'volume': 6000, 'rank': 25},
]
```

```
for k in keywords:
    k['opp'] = round(k['volume'] * k['rank'] / 100, 1) if k['rank'] > 10 else 0
ranked = sorted(keywords, key=lambda k: -k['opp'])
for k in ranked:
    print(f"opp={k['opp']:>7}  rank {k['rank']:>2}  {k['keyword']}")
```

```
output
opp= 1500.0  rank 25  keyword research
opp= 1440.0  rank 18  seo tools
opp=      0  rank  4  free seo audit
```

How it works:

1. A list of dicts holds keyword, volume, and current rank.
2. Each gets an opp key: volume times rank over 100 when rank exceeds 10, else 0.
3. sorted orders the list by negative opp so the largest opportunity leads.
4. Each line prints the opportunity, the rank, and the keyword.

Try it:

1. Treat ranks 11 to 20 as the striking-distance sweet spot and weight them higher.
2. Add a difficulty field and divide the opportunity by it.
3. Print a clear 'page-one already' note for the zero-scored keywords.
4. Change the page-one cutoff from 10 to 3 and re-rank.

⚠ Common Mistakes

- Rewarding rank 1 with a high score chases keywords you have already won.
- Using `>= 10` instead of `> 10` would zero out a keyword sitting exactly at rank 10.
- Sorting ascending buries the biggest opportunity at the bottom.

Skills gained opportunity scoring | conditional formula | descending sort

→ Where You'll Use This: Striking-distance reports lean on exactly this logic to spotlight near-miss keywords where a small ranking push converts into outsized traffic gains.

Mental Model

Like a sprinter in fourth place with the longest stride: not yet on the podium, but the closest to a medal if they push, so coach them first.

Project 201 Estimate Traffic from a CTR-by-Position Table

The Idea Behind It

Ranking position drives click-through rate, so a position-to-CTR table converts rankings into a traffic forecast. The code looks up each keyword's position in the ctr dict, multiplies volume by that rate, rounds to whole clicks, and accumulates a running total. Position 1 pulls 28 percent of 12000 searches for 3360 clicks, and the loop prints each estimate before reporting the summed total.

What You'll Build & Learn

Estimate monthly clicks per keyword and overall by combining volume with a CTR-by-position table.

- CTR varies sharply by SERP position
- Dict lookup of position to rate
- Accumulating a running total in a loop

Skill Roadmap

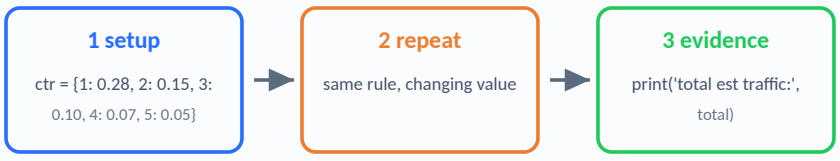
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 200: score opportunity from volume and current rank. Project 202 changes the surface problem while preserving the underlying method. *progress check*

Reading target: estimate traffic from a ctr-by-position table is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print('total est traffic:', total)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for kw, vol, pos in rows:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('total est traffic:', total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for kw, vol, pos in rows:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Estimate Traffic from a CTR-by-Position Table: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: ctr, rows, seo, guide, keyword. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05} | Core: for kw, vol, pos in rows: | Evidence: print('total est traffic:', total)

🛠️ Reader to Writer Bridge

1. Read: explain estimate traffic from a ctr-by-position table in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 200.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 202 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('total est traffic:', total)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of estimate traffic from a ctr-by-position table and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Looking up each ranking's CTR and multiplying by volume yields per-keyword estimated clicks, which sum to a total traffic projection.

```
ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}
rows = [('seo guide', 12000, 1), ('keyword tool', 8000, 3), ('rank tracker',
5000, 5)]
total = 0
for kw, vol, pos in rows:
    traffic = round(vol * ctr[pos])
    total += traffic
    print(f'pos {pos}  ctr {ctr[pos]:.0%}  est {traffic:>5}  {kw}')
print('total est traffic:', total)
```

```
output
pos 1  ctr 28%  est  3360  seo guide
pos 3  ctr 10%  est   800  keyword tool
```

```
pos 5  ctr 5%  est   250  rank tracker
total est traffic: 4410
```

How it works:

1. The ctr dict maps positions one through five to click-through rates.
2. rows holds tuples of keyword, volume, and position, and total starts at zero.
3. The loop multiplies volume by the looked-up CTR, rounds it, and adds it to total.
4. Each keyword prints with its position, CTR, and estimate, then the grand total prints.

Try it:

1. Add a default CTR for positions beyond five using dict.get.
2. Apply a brand-vs-generic CTR adjustment per keyword.
3. Report the percentage each keyword contributes to total traffic.
4. Add position 6-10 to the CTR table with smaller values.

⚠ Common Mistakes

- Looking up a position not in the table (e.g. 6) raises KeyError without a `.get` fallback.
- Multiplying volume by a CTR stored as a percent (28) not a fraction (0.28) inflates traffic 100x.
- Forgetting `round` leaves long floats in the estimated click counts.

Skills gained traffic forecasting | CTR table lookup | running total

→ Where You'll Use This: CTR curves turn rank data into projected sessions, the language executives understand, so traffic forecasts justify SEO investment in business terms.

Mental Model

Like estimating a shop's sales from foot traffic and a conversion rate: visitors times the chance each buys gives a revenue forecast before the month even starts.

Project 202 Extract Question Keywords

The Idea Behind It

Keywords that open with who, what, why, or how are explicit questions, prime targets for FAQ content and featured snippets. A list comprehension keeps every keyword whose first split word sits in a question-word tuple. Checking only the first token cleanly isolates true questions from phrases that merely contain a question word elsewhere, and the code prints the count then the sorted matches.

What You'll Build & Learn

Extract question keywords by testing whether the first word is an interrogative.

- Questions start with interrogative words
- Filtering by the first token only
- List comprehension with a membership test

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 201: estimate traffic from a ctr-by-position table. Project 203 changes the surface problem while preserving the underlying method.

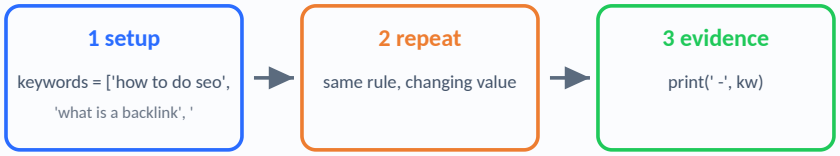
progress check

Reading target: extract question keywords is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = ['how to do seo', 'what is a backlink', 'seo tips', 'why does ranking drop', 'who invented pagerank']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(' -', kw)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = ['how to do seo', 'what is a backlink', 'seo tips', 'why does ranking drop', 'who invented pagerank']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.

Move	Code to inspect	Why it matters
2. Apply the rule	`questions = [kw for kw in keywords if kw.split()[0] in qwords]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' -', kw)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `questions = [kw for kw in keywords if kw.split()[0] in qwords]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Extract Question Keywords: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, how, to, seo, what. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = ['how to do seo', 'what is a backlink', 'seo tips', 'why doe... | Core: questions = [kw for kw in keywords if kw.split()[0] in qwords] | Evidence: print(' -', kw)

Reader to Writer Bridge

1. Read: explain extract question keywords in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 201.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 203 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' –', kw)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `keywords = ['how to do seo', 'what is a backlink', 'seo tips', 'why does ranking drop', 'who invented pagerank']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of extract question keywords and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Testing the first token against a tuple of question words flags interrogative keywords; sorting the matches keeps the printed list stable.

```
keywords = ['how to do seo', 'what is a backlink', 'seo tips', 'why does
ranking drop', 'who invented pagerank']
qwords = ('who', 'what', 'why', 'how', 'when', 'where', 'which')
questions = [kw for kw in keywords if kw.split()[0] in qwords]
print('questions:', len(questions))
for kw in sorted(questions):
    print(' –', kw)
```

```
output
questions: 4
 – how to do seo
 – what is a backlink
 – who invented pagerank
 – why does ranking drop
```

How it works:

1. A list holds five keywords, some questions and some not.
2. qwords is a tuple of interrogatives like who, what, and how.
3. A comprehension keeps each keyword whose kw.split()[0] is in qwords.
4. The code prints how many questions were found, then each sorted question.

Try it:

1. Include keywords ending in a question mark even without a question word.
2. Group the questions by their interrogative word.
3. Add 'can' and 'does' to the interrogative set.
4. Add 'can' and 'does' as question starters and re-run.

⚠ Common Mistakes

- Checking `qword in kw` (substring) flags 'showcase' because it contains 'how'.
- Calling `kw.split()[0]` on an empty string would raise an IndexError.
- Case sensitivity misses a keyword written 'How to do seo' unless you lowercase first.

Skills gained question extraction | first-token check | list comprehension

→ Where You'll Use This: Question keywords feed FAQ schema, People Also Ask targeting, and featured-snippet plays, so isolating them drives a whole content format strategy.

⚡ Mental Model

Like skimming a survey for forms that open with a question mark: the first cue tells you it expects an answer, so you route it to the FAQ desk.

Project 203 Count Modifier Words Across Keywords

💡 The Idea Behind It

Modifiers like best, cheap, free, or 2026 shape both intent and content angle. A Counter tallies how often each modifier appears across the whole keyword list by scanning every word and incrementing when it matches the modifier set. Sorting by descending count then alphabetically shows 'best' leading with two hits, revealing which qualifiers dominate the demand you are working with.

What You'll Build & Learn

Tally modifier words across a keyword list to see which qualifiers dominate.

- Counter for frequency tallies
- Membership filtering inside a loop
- Sorting by count then alphabetically

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 202: extract question keywords. Project 204 changes the surface problem while preserving the underlying method.

progress check

Reading target: count modifier words across keywords is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{mod}: {n}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if w in modifiers:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{mod}: {n}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if w in modifiers:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Modifier Words Across Keywords: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: collections, Counter, keywords, best, seo. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: if w in modifiers: | Evidence: print(f'{mod}: {n}')

🛠️ Reader to Writer Bridge

1. Read: explain count modifier words across keywords in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 202.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 204 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{mod}: {n}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count modifier words across keywords and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Iterating every word and counting hits against a modifier set with a `Counter` reveals the qualifiers your audience appends most often.

```
from collections import Counter
keywords = ['best seo tool', 'cheap seo tool', 'seo tool near me', 'best
keyword tool', 'free seo tool 2026']
modifiers = ('best', 'cheap', 'free', 'top', 'near', 'me', '2026')
found = Counter()
for kw in keywords:
    for w in kw.split():
        if w in modifiers:
            found[w] += 1
for mod, n in sorted(found.items(), key=lambda x: (-x[1], x[0])):
    print(f'{mod}: {n}')
```

```
output
best: 2
2026: 1
cheap: 1
free: 1
me: 1
near: 1
```

How it works:

1. modifiers is a tuple of qualifier words, and found is an empty Counter.
2. Nested loops walk each keyword's words and increment found[w] whenever w is a known modifier.
3. found.items() is sorted by negative count then by modifier name.
4. Each modifier prints with its total occurrences.

Try it:

1. Add commercial modifiers like 'review' and 'compare' to the set.
2. Report each modifier's share as a percentage of all matches.
3. Count two-word modifiers such as 'near me' as a single unit.
4. Add 'review' and 'vs' to the modifier set.

⚠ Common Mistakes

- Treating 'near me' as one token misses both 'near' and 'me' that split apart.
- Sorting only by count leaves equal counts in nondeterministic order without the name tiebreaker.
- Putting '2026' in the set as an int (2026) never matches the string token '2026'.

Skills gained modifier analysis | Counter | frequency sort

→ Where You'll Use This: Modifier frequency exposes whether your keyword set skews toward price-shoppers, deal-seekers, or year-specific queries, shaping headlines and meta copy.

⚡ Mental Model

Like counting adjectives in a stack of restaurant reviews: if 'cheap' keeps surfacing, you learn your audience cares more about price than ambience.

Project 204 Split Branded from Non-Branded Keywords

💡 The Idea Behind It

Branded keywords contain your brand name and convert differently from generic discovery terms, so reports separate the two. Two list comprehensions split the keywords: one keeps phrases where the brand is a whole word in the split, the other keeps those where it is not. The whole-word check via split avoids matching the brand inside an unrelated longer word, and counts print with each group.

What You'll Build & Learn

Split a keyword list into branded and non-branded buckets using a whole-word brand check.

- Branded versus generic keyword behaviour
- Whole-word matching via split membership
- Two complementary comprehensions

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 203: count modifier words across keywords. Project 205 changes the surface problem while preserving the underlying method.

progress check

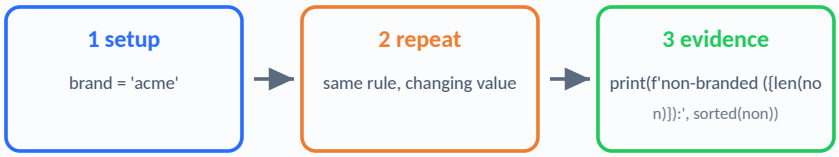
Reading target: split branded from non-branded keywords is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `brand = 'acme'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'non-branded ({len(non)}):', sorted(non))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`brand = 'acme'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`branded = [k for k in keywords if brand in k.split()]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'non-branded ({len(non)}):', sorted(non))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `branded = [k for k in keywords if brand in k.split()]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Split Branded from Non-Branded Keywords: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: brand, acme, keywords, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: brand = 'acme' | Core: branded = [k for k in keywords if brand in k.split()] | Evidence: print(f'non-branded ({len(non)}):', sorted(non))

🛠️ Reader to Writer Bridge

1. Read: explain split branded from non-branded keywords in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 203.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 205 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'non-branded ({len(non)}):', sorted(non))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `brand = 'acme'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split branded from non-branded keywords and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Testing `brand in k.split()` matches the brand as a whole word, partitioning keywords into branded and non-branded buckets for separate analysis.

```
brand = 'acme'
keywords = ['acme running shoes', 'best running shoes', 'acme login', 'cheap
trainers', 'acme reviews']
branded = [k for k in keywords if brand in k.split()]
non = [k for k in keywords if brand not in k.split()]
print(f'branded ({len(branded)}):', sorted(branded))
print(f'non-branded ({len(non)}):', sorted(non))
```

```
output
branded (3): ['acme login', 'acme reviews', 'acme running shoes']
non-branded (2): ['best running shoes', 'cheap trainers']
```

How it works:

1. brand is set to 'acme' and keywords holds five mixed phrases.
2. branded keeps each keyword where brand is in kw.split(), a whole-word test.
3. non keeps the complement where brand is absent from the words.
4. Both groups print with their counts and sorted keywords.

Try it:

1. Support a list of brand aliases instead of a single string.
2. Make the brand match case-insensitive.
3. Report the share of total volume that is branded versus generic.
4. Lowercase keywords first so 'ACME store' counts as branded.

⚠ Common Mistakes

- Using `brand in k` (substring) would match a brand fragment inside an unrelated word.
- Case sensitivity misses 'Acme Login' unless both sides are lowercased.
- A multi-word brand like 'acme corp' will not match a single-token membership test.

Skills gained `brand segmentation` `whole-word matching` `list comprehension`

→ **Where You'll Use This:** Separating branded from non-branded keywords is standard in performance reporting because brand traffic and generic discovery deserve different goals and budgets.

⚡ Mental Model

Like separating mail addressed to you by name from flyers addressed 'to the resident': the named ones already know who you are, so they behave differently.

Project 205 Map Keywords to Funnel Stages

💡 The Idea Behind It

Awareness, consideration, and decision keywords each need a different page, so a stage function maps trigger words to a funnel position. It builds a set of the keyword's words and tests for intersection with stage-specific word sets, returning awareness, consideration, or decision in priority order. Using set intersection means any matching trigger fires the stage, and unmatched keywords fall through to unknown.

What You'll Build & Learn

Map each keyword to a marketing-funnel stage using stage-specific trigger words.

- Funnel stages from trigger vocabulary
- Set intersection to test membership
- Ordered checks as stage priority

🧭 Skill Roadmap

`Skill step:` moves you from recognition toward independent tracing. It builds on Project 204: split branded from non-branded keywords. Project 206 changes the surface problem while preserving the underlying method. *progress check*

`Reading target:` map keywords to funnel stages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `keywords = ['what is seo', 'best seo tools', 'seo software pricing', 'buy seo plan']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- `Writing move:` this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.

- Evidence: `print(f'{stage(kw):14} {kw}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = ['what is seo', 'best seo tools', 'seo software pricing', 'buy seo plan']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if w & {'what', 'how', 'guide', 'is'}: return 'awareness'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{stage(kw):14} {kw}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if w & {'what', 'how', 'guide', 'is'}: return 'awareness'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Map Keywords to Funnel Stages: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, what, is, seo, best. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: keywords = ['what is seo', 'best seo tools', 'seo software pricing', 'b... | Core: if w & {'what', 'how', 'guide', 'is'}: return 'awareness' | Evidence: print(f'{stage(kw):14} {kw}')

🛠️ Reader to Writer Bridge

1. Read: explain map keywords to funnel stages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 204.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 206 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{stage(kw):14} {kw}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = ['what is seo', 'best seo tools', 'seo software pricing', 'buy seo plan']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of map keywords to funnel stages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Intersecting each keyword's word set with stage-specific trigger sets routes it to the funnel stage whose triggers it matches first.

```
keywords = ['what is seo', 'best seo tools', 'seo software pricing', 'buy seo
plan']
def stage(kw):
    w = set(kw.split())
    if w & {'what', 'how', 'guide', 'is'}: return 'awareness'
    if w & {'best', 'top', 'review', 'vs'}: return 'consideration'
    if w & {'buy', 'price', 'pricing', 'plan', 'demo'}: return 'decision'
    return 'unknown'
for kw in keywords:
    print(f'{stage(kw):14} {kw}')
```

```
output
awareness      what is seo
consideration  best seo tools
decision       seo software pricing
decision       buy seo plan
```

How it works:

1. A list holds four keywords spanning the funnel.
2. stage turns each keyword into a set of words.
3. It returns awareness, consideration, or decision when that word set intersects the matching trigger set, checked in order.
4. The loop prints each padded stage beside its keyword.

Try it:

1. Add a retention stage with words like 'cancel' and 'support'.
2. Return the matched trigger word alongside the stage for transparency.
3. Count how many keywords land in each funnel stage.
4. Add a 'retention' stage for 'support' and 'renew' keywords.

⚠ Common Mistakes

- Set intersection on a string instead of `kw.split()` compares characters, not words.
- Order of checks matters: an awareness trigger fires before a later decision one.
- A keyword matching no stage silently returns 'unknown' with no warning.

Skills gained funnel mapping | set intersection | intent staging

→ Where You'll Use This: Funnel mapping tells content teams whether a keyword wants a top-of-funnel guide, a mid-funnel comparison, or a bottom-of-funnel pricing page.

Mental Model

Like a shop greeter reading body language: a browser gets a brochure, a comparer gets a demo, and a wallet-out customer gets the checkout, each at the right moment.

Project 206 Dedupe Near-Identical Keywords by Canonical Form

The Idea Behind It

Keyword exports are full of cosmetic duplicates: different case, extra spaces, reordered words. A canon function lowercases each keyword, splits it, sorts the words, and rejoins them into a canonical form. A seen set tracks which canonical forms have appeared, so only the first variant of each survives in unique. Five noisy inputs collapse to two genuine keywords, with the original spelling preserved.

What You'll Build & Learn

Deduplicate keywords that differ only in case, spacing, or word order via a canonical form.

- Canonicalising for fuzzy deduplication
- Sorting words to ignore order
- A seen set to keep first occurrences

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 205: map keywords to funnel stages. Project 207 changes the surface problem while preserving the underlying method.

progress check

Reading target: dedupe near-identical keywords by canonical form is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

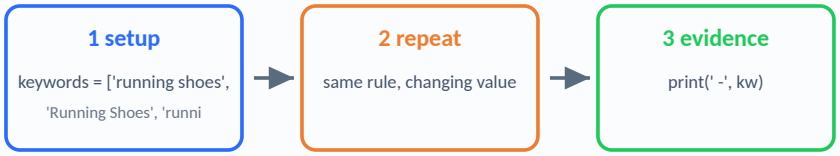
- Setup: `keywords = ['running shoes', 'Running Shoes', 'running shoes', 'shoes running', 'trail shoes']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(' –', kw)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = ['running shoes', 'Running Shoes', 'running shoes', 'shoes running', 'trail shoes']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if c not in seen:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(' –', kw)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if c not in seen:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Dedupe Near-Identical Keywords by Canonical Form: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, running, shoes, Running, Shoes. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: keywords = ['running shoes', 'Running Shoes', 'running shoes', 'shoes r... | Core: if c not in seen: | Evidence: print(' -', kw)

🛠️ Reader to Writer Bridge

1. Read: explain dedupe near-identical keywords by canonical form in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 205.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 207 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' -', kw)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = ['running shoes', 'Running Shoes', 'running shoes', 'shoes running', 'trail shoes']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dedupe near-identical keywords by canonical form and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Lowercasing, splitting, sorting the words, and rejoining produces a canonical key that ignores case, spacing, and word order, so a `seen` set keeps only the first of each variant.

```
keywords = ['running shoes', 'Running Shoes', 'running  shoes', 'shoes
running', 'trail shoes']
def canon(kw):
    return ' '.join(sorted(kw.lower().split()))
seen, unique = set(), []
```

```
    for kw in keywords:
        c = canon(kw)
        if c not in seen:
            seen.add(c)
            unique.append(kw)
    print('input:', len(keywords), 'unique:', len(unique))
    for kw in unique:
        print(' -', kw)
```

```
output
input: 5 unique: 2
 - running shoes
 - trail shoes
```

How it works:

1. canon lowercases a keyword, splits it, sorts the words, and joins them with single spaces.
2. seen is an empty set and unique an empty list.
3. Each keyword's canonical form is checked against seen; new forms are recorded and the original keyword kept.
4. The code prints the input and unique counts, then each surviving keyword.

Try it:

1. Strip punctuation inside canon so 'shoes,' matches 'shoes'.
2. Keep the variant with the highest volume instead of the first seen.
3. Report which originals collapsed into each canonical form.
4. Drop the word-sort step and see 'shoes running' survive as distinct.

⚠ Common Mistakes

- Comparing raw strings misses 'Running Shoes' and 'shoes running' as duplicates.
- Sorting words treats 'dog food' and 'food dog' as the same, which may be wrong for ordered phrases.
- Splitting on `kw.split(' ')` keeps empty tokens from double spaces; bare `.split()` collapses them.

Skills gained keyword deduplication | canonical form | set tracking

→ Where You'll Use This: Canonical dedupe cleans messy multi-tool keyword exports before analysis, stopping the same query from being counted three times under different spellings.

Mental Model

Like alphabetising the letters of each word to spot anagrams: 'shoes running' and 'running shoes' resolve to the same fingerprint, so you keep just one.

Project 207 Rank Keywords by Estimated Commercial Value

The Idea Behind It

Volume alone ignores money, so estimated value multiplies volume by cost-per-click and conversion rate to surface keywords worth the most revenue rather than the most clicks. The loop writes a value key into each dict, and sorting by negative value ranks them. 'enterprise seo', low in volume but high in CPC and conversion, beats a 12000-volume term, proving that traffic and revenue are not the same.

What You'll Build & Learn

Rank keywords by estimated commercial value from volume, cost-per-click, and conversion rate.

- Commercial value beats raw volume
- A three-factor revenue proxy formula
- Sorting dicts by a computed field

Skill Roadmap

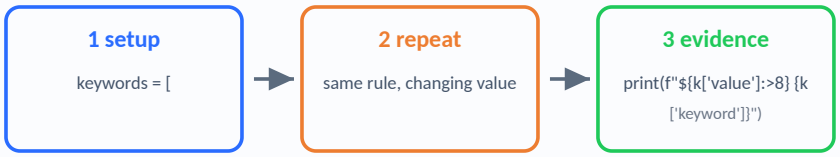
Skill step: moves you from recognition toward independent tracing. It builds on Project 206: dedupe near-identical keywords by canonical form. Project 208 changes the surface problem while preserving the underlying method. *progress check*

Reading target: rank keywords by estimated commercial value is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f"${k['value']:>8} {k['keyword']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for k in keywords:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"${k['value']:>8} {k['keyword']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for k in keywords:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Rank Keywords by Estimated Commercial Value: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, seo, tool, volume. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = [| Core: for k in keywords: | Evidence: print(f"${k['value']:>8} {k['keyword']}")

Reader to Writer Bridge

1. Read: explain rank keywords by estimated commercial value in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 206.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 208 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"${k['value']:>8} {k['keyword']}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of rank keywords by estimated commercial value and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

From Concept to Code (Python)

Multiplying volume by CPC by conversion rate weights each keyword by likely revenue, so a low-volume but high-CPC, high-converting term can outrank a popular cheap one.

```
keywords = [
    {'keyword': 'seo tool', 'volume': 8000, 'cpc': 4.5, 'conv': 0.02},
    {'keyword': 'free seo', 'volume': 12000, 'cpc': 0.5, 'conv': 0.01},
    {'keyword': 'enterprise seo', 'volume': 1500, 'cpc': 12.0, 'conv': 0.05},
]
for k in keywords:
    k['value'] = round(k['volume'] * k['cpc'] * k['conv'], 1)
```

```
for k in sorted(keywords, key=lambda k: -k['value']):
    print(f"${k['value']:>8}  {k['keyword']}")
```

```
output
$   900.0  enterprise seo
$   720.0  seo tool
$    60.0  free seo
```

How it works:

1. A list of dicts holds keyword, volume, cpc, and conv (conversion rate).
2. The loop computes value as volume times cpc times conv, rounded, into each dict.
3. sorted orders the dicts by negative value, most valuable first.
4. Each line prints a dollar-prefixed value beside the keyword.

Try it:

1. Add a difficulty field and divide value by it for value-per-effort.
2. Print the dollar values formatted to two decimal places.
3. Multiply by an average order value instead of a flat conversion rate.
4. Add a 'margin' factor and multiply it into the value.

⚠ Common Mistakes

- Ranking by volume alone would put 'free seo' on top despite its tiny value.
- Storing conversion as a percent (2) not a fraction (0.02) inflates value 100x.
- Sorting ascending lists the least valuable keyword first.

Skills gained value scoring | revenue proxy | descending sort

→ Where You'll Use This: Value-based ranking aligns keyword strategy with revenue, helping you defend why a low-traffic high-intent term outranks a vanity keyword in the roadmap.

⚡ Mental Model

Like a fishing spot with few but huge fish beating a crowded pond of minnows: you weigh the catch's worth, not just how many bites you get.

Project 208 Map Keywords to Target URLs

💡 The Idea Behind It

Each keyword should have one page that owns it, so a keyword-to-URL map lets you check coverage and spot orphans. A mapping dict pairs keywords with their target URLs, and the code looks up each query with dict.get,

supplying a 'NO TARGET PAGE' fallback. Matched queries print their URL; the unmapped 'climbing shoes' prints the fallback, instantly revealing a content gap.

What You'll Build & Learn

Match queries to their owning URL and flag any keyword with no target page.

- One keyword maps to one owning page
- dict.get with a default fallback
- Detecting coverage gaps

🧭 Skill Roadmap

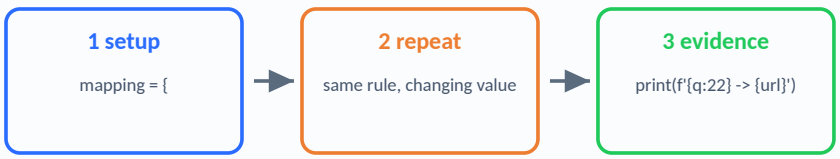
Skill step: moves you from recognition toward independent tracing. It builds on Project 207: rank keywords by estimated commercial value. Project 209 changes the surface problem while preserving the underlying method. *progress check*

Reading target: map keywords to target urls is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `mapping = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{q:22} -> {url}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`mapping = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for q in queries:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{q:22} -> {url}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for q in queries:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Map Keywords to Target URLs: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: mapping, best, running, shoes, blog. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: mapping = { | Core: for q in queries: | Evidence: print(f'{q:22} -> {url}')

🛠️ Reader to Writer Bridge

1. Read: explain map keywords to target urls in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 207.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 209 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{q:22} -> {url}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `mapping = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of map keywords to target urls and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A dict lookup with `.get` and a default returns the owning URL or a clear flag, exposing keywords that still need a landing page.

```
mapping = {
    'best running shoes': '/blog/best-running-shoes',
    'running shoes': '/shop/running-shoes',
    'trail shoes': '/shop/trail-shoes',
}
queries = ['running shoes', 'best running shoes', 'climbing shoes']
for q in queries:
    url = mapping.get(q, 'NO TARGET PAGE')
    print(f'{q:22} -> {url}')
```

```
output
running shoes          -> /shop/running-shoes
best running shoes     -> /blog/best-running-shoes
climbing shoes         -> NO TARGET PAGE
```

How it works:

1. The mapping dict pairs three keywords with their target URLs.
2. queries lists the terms to check, including an unmapped one.
3. For each query, dict.get returns the URL or the 'NO TARGET PAGE' default.

4. Each query prints left-padded beside its resolved URL or the gap marker.

Try it:

1. Collect all unmapped queries into a separate gap list.
2. Allow one URL to own multiple keywords and validate the reverse.
3. Load the mapping from a CSV export of your site structure.
4. Collect all queries that returned NO TARGET PAGE into a gap list.

⚠ Common Mistakes

- Using `mapping[q]` instead of `.get` raises KeyError for an unmapped keyword.
- Case or whitespace mismatches between query and map keys cause silent misses.
- Treating the default string as a real URL downstream would break link generation.

Skills gained keyword-to-URL mapping | dict.get default | gap detection

→ Where You'll Use This: Keyword-to-URL maps are the backbone of content gap audits, showing which target terms still lack a dedicated landing page to win them.

⚡ Mental Model

Like a seating chart at a wedding: every guest should have one assigned table, and an unlisted name means someone has nowhere to sit yet.

Project 209 Flag Keywords for SERP-Feature Targeting

💡 The Idea Behind It

Some keywords trigger SERP features like featured snippets you can win with structured content. The code stores a list of SERP features per keyword, then flags any keyword whose feature list contains the chosen target. A simple membership test sets the flag to TARGET or a dash, so two keywords showing featured_snippet get flagged while a shopping-only keyword does not, focusing optimisation effort.

What You'll Build & Learn

Flag keywords whose SERP shows a chosen feature, like featured snippets, for targeting.

- SERP features as winnable opportunities
- Membership test against a feature list
- Conditional flag formatting

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 208: map keywords to target urls. Project 210 changes the surface problem while preserving the underlying method.

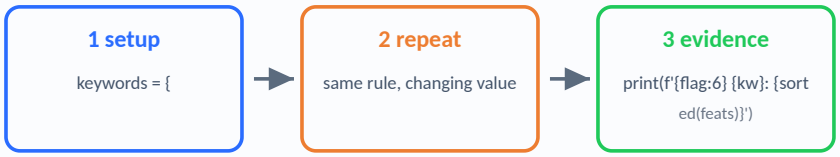
progress check

Reading target: flag keywords for serp-feature targeting is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{flag:6} {kw}: {sorted(feats)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`flag = 'TARGET' if target in feats else '-'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{flag:6} {kw}: {sorted(feats)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = 'TARGET' if target in feats else '–'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Keywords for SERP-Feature Targeting: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, how, to, tie, shoes. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = { | Core: flag = 'TARGET' if target in feats else '-' | Evidence: print(f'{flag:6} {kw}: {sorted(feats)}')

🛠️ Reader to Writer Bridge

1. Read: explain flag keywords for serp-feature targeting in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 208.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 210 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{flag:6} {kw}: {sorted(feats)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag keywords for serp-feature targeting and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Testing whether the target feature is in each keyword's feature list flags the snippet-winnable opportunities, with sorted features for stable output.

```
keywords = {
    'how to tie shoes': ['featured_snippet'],
    'nike shoes': ['shopping', 'image_pack'],
    'shoe size chart': ['featured_snippet', 'people_also_ask'],
}
target = 'featured_snippet'
for kw in sorted(keywords):
    feats = keywords[kw]
    flag = 'TARGET' if target in feats else '-'
    print(f'{flag:6} {kw}: {sorted(feats)}')
```

```
output
TARGET how to tie shoes: ['featured_snippet']
-      nike shoes: ['image_pack', 'shopping']
TARGET shoe size chart: ['featured_snippet', 'people_also_ask']
```

How it works:

1. The keywords dict maps each keyword to a list of SERP features it triggers.
2. target is set to 'featured_snippet'.
3. For each sorted keyword, flag becomes 'TARGET' when target is in its feature list, else a dash.
4. Each line prints the flag, the keyword, and its sorted features.

Try it:

1. Accept a list of target features and flag a match on any of them.
2. Count how many keywords trigger each distinct SERP feature.
3. Sort flagged keywords ahead of unflagged ones in the output.
4. Switch the target to 'people_also_ask' and re-run.

⚠ Common Mistakes

- Checking `target == feats` compares a string to a list and never matches.
- Not sorting the feature lists makes the printed order vary by insertion order.
- Hard-coding one target misses keywords that show other winnable features.

Skills gained SERP feature targeting | list membership | flag formatting

→ Where You'll Use This: SERP-feature flags direct structured-content efforts toward snippets, image packs, or People Also Ask boxes where a format tweak can capture extra visibility.

Mental Model

Like scanning a stage for open mic slots: you only step up where there's a spot you can actually fill, ignoring the segments already locked down.

Project 210 Filter Keywords by a Difficulty-to-Volume Ratio

The Idea Behind It

A volume-to-difficulty ratio ranks bang-for-buck, but a hard difficulty ceiling stops you chasing impossible terms. The loop computes ratio as volume over difficulty, then sorts descending. A verdict flags GO when difficulty is at or below 40, else skip, which is why 'seo' has the top ratio yet still gets skipped at difficulty 90. Combining the ratio with the cap yields a realistic shortlist.

What You'll Build & Learn

Rank keywords by a volume-to-difficulty ratio while capping out terms above a difficulty ceiling.

- Volume-to-difficulty as bang-for-buck
- Why a hard ceiling matters too
- Combining a ratio with a gate

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 209: flag keywords for serp-feature targeting. Project 211 changes the surface problem while preserving the underlying method.

progress check

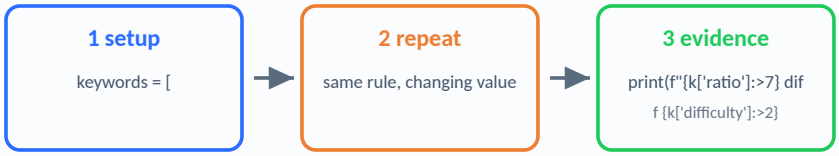
Reading target: filter keywords by a difficulty-to-volume ratio is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f"{k['ratio']:>7} diff {k['difficulty']:>2} {verdict:4} {k['keyword']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`verdict = 'GO' if k['difficulty'] <= 40 else 'skip'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{k['ratio']:>7} diff {k['difficulty']:>2} {verdict:4} {k['keyword']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'GO' if k['difficulty'] <= 40 else 'skip'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Filter Keywords by a Difficulty-to-Volume Ratio: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, seo, basics, volume. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: keywords = [| Core: verdict = 'GO' if k['difficulty'] <= 40 else 'skip' | Evidence: print(f"{k['ratio']:>7} diff {k['difficulty']:>2} {verdict:4} {k['keywo...

🛠️ Reader to Writer Bridge

1. Read: explain filter keywords by a difficulty-to-volume ratio in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 209.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 211 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{k['ratio']:>7} diff {k['difficulty']:>2} {verdict:4} {k['keyword']}")`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of filter keywords by a difficulty-to-volume ratio and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The ratio ranks efficiency while the separate difficulty ceiling vetoes terms that are too hard, so a high-ratio but brutally hard keyword still reads 'skip'.

```
keywords = [
    {'keyword': 'seo basics', 'volume': 5000, 'difficulty': 20},
    {'keyword': 'seo', 'volume': 40000, 'difficulty': 90},
    {'keyword': 'local seo guide', 'volume': 3000, 'difficulty': 25},
]
for k in keywords:
    k['ratio'] = round(k['volume'] / k['difficulty'], 1)
for k in sorted(keywords, key=lambda k: -k['ratio']):
    verdict = 'GO' if k['difficulty'] <= 40 else 'skip'
    print(f"{k['ratio']:>7}  diff {k['difficulty']:>2}  {verdict:4}
{k['keyword']}")
```

```
output
  444.4  diff 90  skip  seo
  250.0  diff 20  GO    seo basics
  120.0  diff 25  GO    local seo guide
```

How it works:

1. A list of dicts holds keyword, volume, and difficulty.
2. The loop writes ratio as volume divided by difficulty, rounded.
3. sorted orders by negative ratio, best value first.
4. Each line's verdict is GO when difficulty is 40 or less, otherwise skip, printed beside the keyword.

Try it:

1. Make the difficulty ceiling a tunable variable.
2. Add a minimum volume floor so tiny terms drop out.
3. Show the GO keywords first regardless of ratio order.
4. Lower the difficulty ceiling to 25 and watch 'local seo guide' flip.

⚠ Common Mistakes

- Relying on ratio alone keeps 'seo' on top despite a difficulty of 90.

- Sorting by volume instead of ratio loses the efficiency view.
- Using `<` instead of `<=` on the ceiling would skip a keyword at exactly 40.

Skills gained ratio scoring | difficulty ceiling | verdict labelling

→ **Where You'll Use This:** Ratio-plus-cap filtering builds a defensible target list that balances efficiency against realism, the judgment call every keyword shortlist must make.

Mental Model

Like shopping for value but refusing anything over a price cap: a luxury car may be great per dollar, yet it still busts the budget, so you skip it.

Project 211 Find Keywords Containing a Seed Term

The Idea Behind It

Keyword expansion starts from a seed: pull every phrase that contains your core word. A list comprehension keeps each keyword where the seed appears as a whole word in the split, so 'running' matches three of five phrases without firing on a substring inside a longer word. The code reports the hit ratio then prints the sorted matches, a tiny but reliable expansion filter.

What You'll Build & Learn

Find every keyword that contains a seed term using a whole-word match.

- Seed-based keyword expansion
- Whole-word matching with split
- Reporting a hit ratio

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 210: filter keywords by a difficulty-to-volume ratio. Project 212 changes the surface problem while preserving the underlying method.

progress check

Reading target: find keywords containing a seed term is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = ['running shoes', 'best running shoes', 'running shoe deals', 'trail hiking', 'wool socks']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(' –', kw)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = ['running shoes', 'best running shoes', 'running shoe deals', 'trail hiking', 'wool socks']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`related = [kw for kw in keywords if seed in kw.split()]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' –', kw)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `related = [kw for kw in keywords if seed in kw.split()]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Keywords Containing a Seed Term: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, running, shoes, best, shoe. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = ['running shoes', 'best running shoes', 'running shoe deals'... | Core: related = [kw for kw in keywords if seed in kw.split()] | Evidence: print(' -', kw)

🛠️ Reader to Writer Bridge

1. Read: explain find keywords containing a seed term in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 210.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 212 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' -', kw)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `keywords = ['running shoes', 'best running shoes', 'running shoe deals', 'trail hiking', 'wool socks']` and say why this line, value, import, or entry

point is needed before the rest of the Python can work.

3. Transfer: change one small part of find keywords containing a seed term and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Filtering by `seed in kw.split()` keeps only phrases that contain the seed as a standalone word, then sorting yields a stable expansion list.

```
keywords = ['running shoes', 'best running shoes', 'running shoe deals', 'trail
hiking', 'wool socks']
seed = 'running'
related = [kw for kw in keywords if seed in kw.split()]
print(f'seed {seed!r} appears in {len(related)} of {len(keywords)}')
for kw in sorted(related):
    print(' -', kw)
```

```
output
seed 'running' appears in 3 of 5
 - best running shoes
 - running shoe deals
 - running shoes
```

How it works:

1. A list holds five keywords and seed is set to 'running'.
2. A comprehension keeps each keyword where seed is in kw.split(), a whole-word test.
3. The code prints how many of the total contain the seed.
4. Each matching keyword prints sorted on its own line.

Try it:

1. Accept several seeds and keep keywords matching any of them.
2. Make the match case-insensitive.
3. Also report the keywords that do not contain the seed.
4. Switch the seed to 'shoes' and compare the matches.

⚠ Common Mistakes

- Using `seed in kw` (substring) would also match 'overrunning' or similar fragments.
- Case mismatch misses 'Running shoes' unless both sides are lowercased.
- Forgetting to sort makes the printed expansion order depend on input order.

Skills gained seed expansion whole-word match list comprehension

→ Where You'll Use This: Seed filtering powers keyword expansion workflows, isolating every phrase built around a core topic word so you can map a cluster around it.

Mental Model

Like pulling every file folder with a project codeword on the tab: you skim for the exact word, not a chance letter run, so you grab only the right ones.

Project 212 Rank Token Frequency Across a Keyword List

The Idea Behind It

The most common words across a keyword set reveal dominant themes and likely pillar topics. A Counter tallies every token by scanning each keyword's words and incrementing per word, then most_common(4) returns the top four. 'shoes' leads at four and 'running' follows at three, showing how a simple word-frequency tally over the whole list surfaces the themes worth building around.

What You'll Build & Learn

Rank the most frequent tokens across a keyword list to expose dominant themes.

- Token frequency reveals themes
- Counter for tallying
- most_common for top-N results

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 211: find keywords containing a seed term. Project 213 changes the surface problem while preserving the underlying method.

progress check

Reading target: rank token frequency across a keyword list is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.

- Evidence: `print(f'{w}: {n}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for kw in keywords:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{w}: {n}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for kw in keywords:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Rank Token Frequency Across a Keyword List: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: collections, Counter, keywords, best, running. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import Counter | Core: for kw in keywords: | Evidence: print(f'{w}: {n}')

🛠️ Reader to Writer Bridge

1. Read: explain rank token frequency across a keyword list in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 211.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 213 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{w}: {n}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of rank token frequency across a keyword list and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Splitting every keyword into words and tallying with a `Counter` gives a frequency map; `most_common` returns the top tokens, with ties broken by first-seen order.

```
from collections import Counter
keywords = ['best running shoes', 'cheap running shoes', 'running shoes
review', 'trail shoes guide']
counts = Counter()
for kw in keywords:
    for w in kw.split():
        counts[w] += 1
for w, n in counts.most_common(4):
    print(f'{w}: {n}')
```

```
output
shoes: 4
running: 3
best: 1
cheap: 1
```

How it works:

1. counts is an empty Counter and keywords holds four phrases.
2. Nested loops split each keyword and increment counts for every word.
3. counts.most_common(4) returns the four highest-frequency tokens.
4. Each token prints with its total count.

Try it:

1. Exclude stopwords so generic glue words don't dominate.
2. Show the top ten tokens instead of four.
3. Report each token's percentage of all words.
4. Filter out a stop-word list before counting.

⚠ Common Mistakes

- Counting whole keywords instead of their words misses the shared-term theme.
- Stop words like 'for' and 'the' can dominate if you do not filter them out.
- Relying on a precise tie order assumes CPython's insertion-order tiebreak in `most_common`.

Skills gained token frequency | Counter | most_common

→ Where You'll Use This: Word-frequency analysis over a keyword list spots the recurring terms that should anchor pillar pages and inform internal-linking structure.

Mental Model

Like a word cloud rendered as a leaderboard: the words that recur most across your phrases loom largest and point at the topics demand circles around.

Project 213 Pick a Pillar Keyword per Cluster

The Idea Behind It

Each topic cluster needs one pillar page targeting its strongest keyword. A dict comprehension picks a pillar per cluster using max with a key of word count then the keyword itself, so the longest, most descriptive phrase wins, with alphabetical order breaking ties. The output prints each topic, its keyword count, and its chosen pillar, applying one clear heuristic for anchoring a hub page.

What You'll Build & Learn

Select one pillar keyword per cluster by favouring the longest, most descriptive phrase.

- Pillar pages anchor a cluster
- max() with a tuple key
- Word count as a descriptiveness proxy

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 212: rank token frequency across a keyword list. Project 214 changes the surface problem while preserving the underlying method. *progress check*

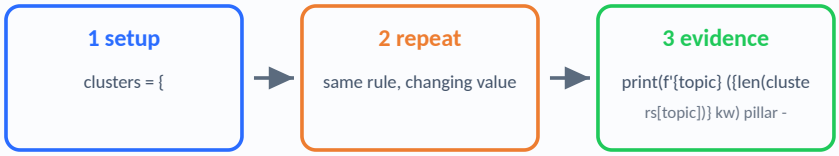
Reading target: pick a pillar keyword per cluster is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `clusters = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{topic} ({len(clusters[topic])} kw) pillar -> {pillars[topic]}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`clusters = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`pillars = {topic: max(kws, key=lambda k: (len(k.split()), k)) for topic, kws in clusters.items()}`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{topic} ({len(clusters[topic])} kw) pillar -> {pillars[topic]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
pillars = {topic: max(kws, key=lambda k: (len(k.split()), k)) for topic, kws in clusters.items()}
```

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Pick a Pillar Keyword per Cluster: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: clusters, shoes, running, trail, best. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: clusters = { | Core: pillars = {topic: max(kws, key=lambda k: (len(k.split()), k)) for topic... | Evidence: print(f'{topic} ({len(clusters[topic])} kw) pillar -> {pillars[topic]}')

🛠️ Reader to Writer Bridge

1. Read: explain pick a pillar keyword per cluster in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 212.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 214 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{topic} ({len(clusters[topic])} kw) pillar -> {pillars[topic]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `clusters = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of pick a pillar keyword per cluster and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`max` with a key of (word count, keyword) picks the longest keyword per cluster, using the keyword string as a deterministic tiebreaker when lengths match.

```
clusters = {
    'shoes': ['running shoes', 'trail shoes', 'best shoes'],
    'socks': ['running socks'],
    'apparel': ['running shorts', 'gym shirt'],
}
pillars = {topic: max(kws, key=lambda k: (len(k.split()), k)) for topic, kws in
clusters.items()}
for topic in sorted(clusters):
    print(f'{topic} ({len(clusters[topic])} kw) pillar -> {pillars[topic]}')
```

```
output
apparel (2 kw) pillar -> running shorts
shoes (3 kw) pillar -> trail shoes
socks (1 kw) pillar -> running socks
```

How it works:

1. The clusters dict maps three topics to lists of keywords.
2. A dict comprehension picks each topic's pillar via max keyed on word count then the keyword.
3. This favours the longest phrase, with the keyword string breaking ties.
4. The loop prints each topic, its keyword count, and the chosen pillar.

Try it:

1. Pick the pillar by highest search volume instead of word count.
2. Require a cluster to have at least two keywords before assigning a pillar.
3. Return the runner-up keyword as a suggested spoke page.
4. Pick the shortest keyword as the pillar instead and compare.

⚠ Common Mistakes

- Picking `max` by raw string length favours long single words over descriptive phrases.
- Omitting the keyword tiebreaker makes equal-length pillars nondeterministic.
- Choosing the longest keyword as a pillar is a heuristic, not always the right SEO call.

Skills gained pillar selection | max with key | dict comprehension

→ Where You'll Use This: Pillar selection is core to hub-and-spoke content architecture, naming the page that ties a cluster's supporting articles together for topical authority.

⚡ Mental Model

Like choosing a team captain: you pick the player whose role spans the most ground, then settle ties by jersey number so the choice is never arbitrary.

Project 214 Score Keywords with Normalized Volume and Difficulty

The Idea Behind It

Volume and difficulty live on different scales, so subtracting them raw is meaningless. A norm function min-max scales a list of values to the zero-to-one range, and the code normalises both volume and difficulty separately. Each keyword's score is normalised volume minus normalised difficulty, zipped back together. 'seo audit' wins on a fair high-volume-minus-difficulty basis once both metrics share a scale.

What You'll Build & Learn

Score keywords on a fair volume-minus-difficulty basis after min-max normalising both metrics.

- Min-max normalisation to a common scale
- Why mixing raw scales is invalid
- zip to recombine parallel lists

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 213: pick a pillar keyword per cluster. Project 215 changes the surface problem while preserving the underlying method.

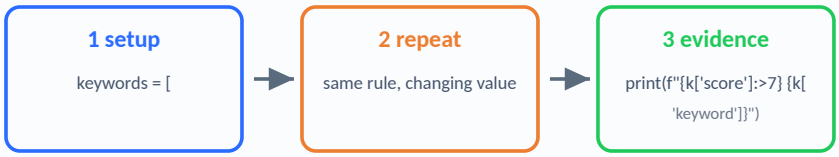
progress check

Reading target: score keywords with normalized volume and difficulty is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f"{k['score']:>7} {k['keyword']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`return [(v - lo) / (hi - lo) for v in vals]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"{k['score']:>7} {k['keyword']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return [(v - lo) / (hi - lo) for v in vals]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Score Keywords with Normalized Volume and Difficulty: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, seo, audit, volume. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = [| Core: return [(v - lo) / (hi - lo) for v in vals] | Evidence: print(f"{k['score']:>7} {k['keyword']}")

🛠️ Reader to Writer Bridge

1. Read: explain score keywords with normalized volume and difficulty in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 213.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 215 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{k['score']:>7} {k['keyword']}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score keywords with normalized volume and difficulty and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Min-max scaling maps each metric to 0-1 so the subtraction `normalised_volume - normalised_difficulty` compares like with like, rewarding high volume and low difficulty equally.

```
keywords = [
    {'keyword': 'seo audit', 'volume': 6000, 'difficulty': 35, 'cpc': 6.0},
    {'keyword': 'free tool', 'volume': 9000, 'difficulty': 70, 'cpc': 1.0},
    {'keyword': 'rank tracker', 'volume': 4000, 'difficulty': 40, 'cpc': 5.0},
]
def norm(vals):
```

```
    lo, hi = min(vals), max(vals)
    return [(v - lo) / (hi - lo) for v in vals]
vols = norm([k['volume'] for k in keywords])
diffs = norm([k['difficulty'] for k in keywords])
for k, nv, nd in zip(keywords, vols, diffs):
    k['score'] = round(nv - nd, 3)
for k in sorted(keywords, key=lambda k: -k['score']):
    print(f"{k['score']:>7}  {k['keyword']}")
```

```
output
    0.4  seo audit
    0.0  free tool
 -0.143  rank tracker
```

How it works:

1. norm min-max scales a list of values into the zero-to-one range.
2. vols and diffs apply norm to the volume and difficulty columns separately.
3. zip pairs each keyword with its normalised volume and difficulty, and score is the difference.
4. sorted by negative score prints the keywords best-first.

Try it:

1. Weight volume more heavily than difficulty in the score.
2. Add a normalised CPC term to the composite score.
3. Guard against a divide-by-zero when all values are equal.
4. Weight volume twice as heavily: `2 * nv - nd`.

⚠ Common Mistakes

- Subtracting raw volume and difficulty lets the larger-scaled metric dominate.
- A constant column (all values equal) makes `hi - lo` zero and divides by zero.
- Relying on `zip` assumes the normalised lists stay in the original keyword order.

Skills gained min-max normalisation | fair scoring | zip

→ Where You'll Use This: Normalised scoring lets you fairly blend metrics on wildly different scales, the foundation of any composite keyword or page-priority index.

Mental Model

Like converting heights and weights to percentiles before comparing athletes: only on a shared zero-to-one ruler does subtracting one from the other mean anything.

Project 215 Summarize Intent Distribution Across a List

The Idea Behind It

A quick portfolio check is the share of keywords in each intent class. The code keeps three counters and, for each keyword, tests its word set against transactional, commercial, then informational trigger sets in order, defaulting to transactional when nothing matches. Dividing each count by the total gives a percentage, so a list skewing 67 percent transactional flags whether the content mix fits the brand's goal.

What You'll Build & Learn

Summarise what share of a keyword list falls into each intent class.

- Intent distribution as a portfolio view
- Counters keyed by intent class
- Computing each class's percentage

Skill Roadmap

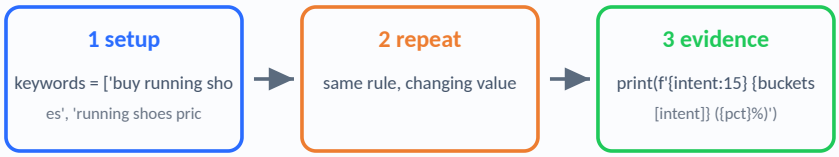
Skill step: pushes the same idea into a less automatic situation. It builds on Project 214: score keywords with normalized volume and difficulty. Project 216 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarize intent distribution across a list is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = ['buy running shoes', 'running shoes price', 'cheap running shoes', 'what are running shoes', 'best running shoes', 'running shoes near me']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{intent:15} {buckets[intent]} ({pct}%)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = ['buy running shoes', 'running shoes price', 'cheap running shoes', 'what are running shoes', 'best running shoes', 'running shoes near me']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if w & {'buy', 'price', 'cheap', 'order'}:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{intent:15} {buckets[intent]} ({pct}%)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if w & {'buy', 'price', 'cheap', 'order'}:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Summarize Intent Distribution Across a List: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, buy, running, shoes, price. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: keywords = ['buy running shoes', 'running shoes price', 'cheap running... | Core: if w & {'buy', 'price', 'cheap', 'order'}: | Evidence: print(f'{intent:15} {buckets[intent]} ({pct}%)')

🛠️ Reader to Writer Bridge

1. Read: explain summarize intent distribution across a list in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 214.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 216 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{intent:15} {buckets[intent]} ({pct}%)')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `keywords = ['buy running shoes', 'running shoes price', 'cheap running shoes', 'what are running shoes', 'best running shoes', 'running shoes near me']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of summarize intent distribution across a list and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Tallying each keyword's intent into a fixed-key dict and converting counts to percentages gives the portfolio mix at a glance.

```
keywords = ['buy running shoes', 'running shoes price', 'cheap running shoes',
'what are running shoes', 'best running shoes', 'running shoes near me']
buckets = {'informational': 0, 'commercial': 0, 'transactional': 0}
```

```
    for kw in keywords:
        w = set(kw.split())
        if w & {'buy', 'price', 'cheap', 'order'}:
            buckets['transactional'] += 1
        elif w & {'best', 'top', 'review', 'vs'}:
            buckets['commercial'] += 1
        elif w & {'what', 'how', 'why', 'guide'}:
            buckets['informational'] += 1
        else:
            buckets['transactional'] += 1
    total = len(keywords)
    for intent in sorted(buckets):
        pct = round(100 * buckets[intent] / total)
        print(f'{intent:15} {buckets[intent]} ({pct}%)')
```

```
output
commercial      1 (17%)
informational   1 (17%)
transactional   4 (67%)
```

How it works:

1. buckets starts three intent counts at zero.
2. For each keyword, its word set is tested against transactional, then commercial, then informational trigger sets, with transactional as the fallback.
3. The matching bucket increments, and total is the keyword count.
4. Each intent prints with its count and rounded percentage.

Try it:

1. Add a navigational bucket and a brand-word trigger for it.
2. Print the distribution sorted by share, largest first.
3. Return the counts as a dict for charting elsewhere.
4. Add a 'navigational' bucket and a brand-name trigger.

⚠ Common Mistakes

- Routing unmatched keywords to transactional inflates that bucket; choose the default deliberately.
- Checking commercial before transactional flips the order and 'cheap' could be miscounted.
- Dividing integer counts before multiplying by 100 truncates the percentage to zero.

Skills gained intent distribution | set intersection | percentage reporting

→ Where You'll Use This: Intent-mix summaries reveal whether a keyword portfolio matches business goals, catching a commerce site that has accidentally over-indexed on informational terms.

Mental Model

Like checking the pie-chart split of a stock portfolio: if 70 percent sits in one sector when you wanted balance, the imbalance jumps out at a glance.

Project 216 Find Quick-Win Keywords by Volume and Difficulty

The Idea Behind It

Quick wins are keywords with real volume but low difficulty, the ones you can rank for soon. A list comprehension keeps keywords passing two conditions at once: difficulty under 30 and volume at least 1000. Sorting the survivors by negative volume prints the biggest quick win first, and a final count confirms how many made the cut from the larger list.

What You'll Build & Learn

Pull quick-win keywords that clear both a volume floor and a difficulty ceiling.

- Quick wins combine demand and ease
- Two-condition filter with and
- Sorting survivors by volume

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 215: summarize intent distribution across a list. Project 217 changes the surface problem while preserving the underlying method.

progress check

Reading target: find quick-win keywords by volume and difficulty is the surface task; the transferable Python move is to mark setup, rule, and output before running.

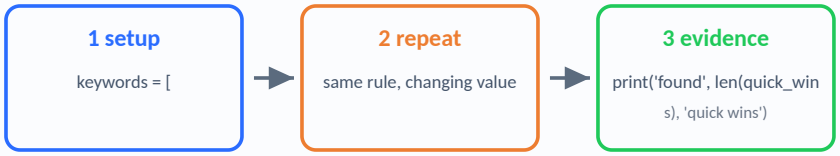
setup -> rule -> result

Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print('found', len(quick_wins), 'quick wins')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`quick_wins = [k for k in keywords if k['difficulty'] < 30 and k['volume'] >= 1000]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('found', len(quick_wins), 'quick wins')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `quick_wins = [k for k in keywords if k['difficulty'] < 30 and k['volume'] >= 1000]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Quick-Win Keywords by Volume and Difficulty: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, seo, guide, volume. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = [| Core: quick_wins = [k for k in keywords if k['difficulty'] < 30 and k['volume... | Evidence: print('found', len(quick_wins), 'quick wins')

🛠️ Reader to Writer Bridge

1. Read: explain find quick-win keywords by volume and difficulty in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 215.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 217 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('found', len(quick_wins), 'quick wins')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find quick-win keywords by volume and difficulty and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A single comprehension applying both a difficulty ceiling and a volume floor isolates the easy, worthwhile keywords; sorting by volume orders the best first.

```
keywords = [
    {'keyword': 'seo guide', 'volume': 5000, 'difficulty': 22},
    {'keyword': 'link building', 'volume': 4000, 'difficulty': 55},
    {'keyword': 'meta tags', 'volume': 3000, 'difficulty': 18},
    {'keyword': 'serp features', 'volume': 800, 'difficulty': 40},
```

```
]
quick_wins = [k for k in keywords if k['difficulty'] < 30 and k['volume'] >=
1000]
for k in sorted(quick_wins, key=lambda k: -k['volume']):
    print(f"QUICK WIN: {k['keyword']} (vol {k['volume']}, diff
{k['difficulty']})")
print('found', len(quick_wins), 'quick wins')
```

```
output
QUICK WIN: seo guide (vol 5000, diff 22)
QUICK WIN: meta tags (vol 3000, diff 18)
found 2 quick wins
```

How it works:

1. A list of dicts holds keyword, volume, and difficulty.
2. quick_wins keeps keywords where difficulty is under 30 and volume is at least 1000.
3. Survivors sort by negative volume so the biggest prints first.
4. Each prints a QUICK WIN line, then the total count of quick wins prints.

Try it:

1. Make both thresholds adjustable function parameters.
2. Add a third condition requiring transactional or commercial intent.
3. Print the count of keywords that failed the filter too.
4. Loosen the difficulty ceiling to 40 and recount.

⚠ Common Mistakes

- Using `or` instead of `and` admits high-volume but brutally hard keywords.
- A volume floor of 0 lets near-zero-traffic terms count as wins.
- Forgetting the sort leaves the wins in arbitrary input order.

Skills gained quick-win filtering | compound condition | volume sort

→ Where You'll Use This: Quick-win lists give content teams an immediate sprint backlog: low-difficulty, decent-volume keywords that deliver early ranking momentum and stakeholder confidence.

⚡ Mental Model

Like picking low-hanging fruit that is also ripe: you want the apples both within reach and worth eating, not the green ones at the top of the tree.

Project 217 Detect Near-Duplicate Keywords with String Similarity

The Idea Behind It

Typos and singular-plural variants bloat keyword lists in ways exact-match dedupe misses. The code uses difflib.SequenceMatcher to compute a character-level similarity ratio between a seed and each candidate, flagging NEAR-DUP at a 0.85 threshold. 'running shoe' and the misspelled 'runing shoes' both score 0.96 and get flagged, while genuinely different phrases stay distinct, catching fuzzy duplicates a set could never see.

What You'll Build & Learn

Flag near-duplicate keywords against a seed using a string-similarity ratio.

- String similarity catches fuzzy dupes
- difflib.SequenceMatcher ratio
- A similarity threshold for flagging

Skill Roadmap

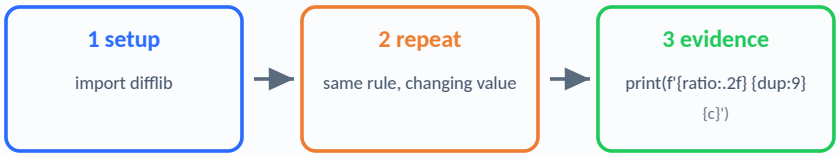
Skill step: pushes the same idea into a less automatic situation. It builds on Project 216: find quick-win keywords by volume and difficulty. Project 218 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect near-duplicate keywords with string similarity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import difflib` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{ratio:.2f} {dup:9} {c}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import difflib`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`dup = 'NEAR-DUP' if ratio >= 0.85 else 'distinct'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{ratio:.2f} {dup:9} {c}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dup = 'NEAR-DUP' if ratio >= 0.85 else 'distinct'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Near-Duplicate Keywords with String Similarity: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: difflib, seed, running, shoes, candidates. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import difflib | Core: dup = 'NEAR-DUP' if ratio >= 0.85 else 'distinct' | Evidence: print(f'{ratio:.2f} {dup:9} {c}')

🛠️ Reader to Writer Bridge

1. Read: explain detect near-duplicate keywords with string similarity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 216.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 218 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{ratio:.2f} {dup:9} {c}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import difflib` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect near-duplicate keywords with string similarity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

`difflib.SequenceMatcher.ratio` returns a 0-1 character-overlap score; a high threshold catches typos and singular/plural variants that exact comparison treats as different keywords.

```
import difflib
seed = 'running shoes'
candidates = ['running shoe', 'runing shoes', 'walking shoes', 'running shoes',
'trail shoes']
for c in candidates:
    ratio = difflib.SequenceMatcher(None, seed, c).ratio()
```

```
    dup = 'NEAR-DUP' if ratio >= 0.85 else 'distinct'
    print(f'{ratio:.2f}  {dup:9} {c}')
```

```
output
0.96  NEAR-DUP  running shoe
0.96  NEAR-DUP  runing shoes
0.69  distinct  walking shoes
1.00  NEAR-DUP  running shoes
0.67  distinct  trail shoes
```

How it works:

1. seed is 'running shoes' and candidates lists five variants and unrelated terms.
2. For each candidate, SequenceMatcher(None, seed, c).ratio() gives a similarity between zero and one.
3. dup becomes NEAR-DUP at a ratio of 0.85 or higher, else distinct.
4. Each line prints the ratio, the verdict, and the candidate.

Try it:

1. Compare every pair in a list, not just against one seed.
2. Lower the threshold and observe which distinct terms get caught.
3. Group flagged near-duplicates with the seed they matched.
4. Raise the threshold to 0.95 and watch 'running shoe' become distinct.

⚠ Common Mistakes

- Setting the threshold too low (e.g. 0.5) merges genuinely different keywords like 'walking shoes'.
- SequenceMatcher is character-based, so it can score reordered phrases lower than expected.
- Comparing without lowercasing lets case differences drag the ratio down.

Skills gained fuzzy deduplication | difflib similarity | thresholding

→ Where You'll Use This: Fuzzy similarity catches the typo and plural variants that exact dedupe leaves behind, tightening keyword lists before clustering or reporting.

Mental Model

Like a spell-checker that underlines words almost but not quite right: it scores how close the letters line up and warns you when two phrases are suspiciously alike.

Project 218 Compare Two Keywords by SERP Overlap

The Idea Behind It

Two keywords with the same ranking pages probably share search intent and should target one page. The code stores top domains per keyword, takes the intersection and union of two keywords' domain sets, and divides their sizes for a SERP-overlap similarity. At 0.5 overlap the verdict reads 'same intent', so a domain-overlap measure decides whether to merge or separate two terms.

What You'll Build & Learn

Compare two keywords by SERP domain overlap to judge whether they share intent.

- Shared ranking pages signal shared intent
- Set overlap of SERP domains
- An inline verdict via conditional expression

🧭 Skill Roadmap

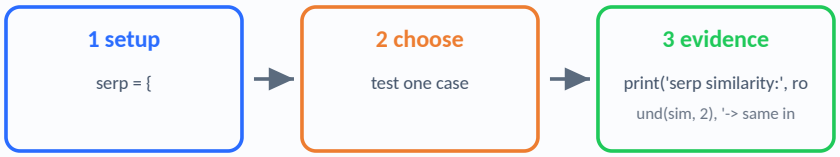
Skill step: pushes the same idea into a less automatic situation. It builds on Project 217: detect near-duplicate keywords with string similarity. Project 219 changes the surface problem while preserving the underlying method. *progress check*

Reading target: compare two keywords by serp overlap is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `serp = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4 else '-> different')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`serp = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4 else '-> different')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. slow line
3. Check the evidence	`print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4 else '-> different')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE
```
print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4 else '-> different')
```

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Compare Two Keywords by SERP Overlap: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: serp, seo, tools, ahrefs, com. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: serp = { | Core: print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4... | Evidence: print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4...

🛠️ Reader to Writer Bridge

1. Read: explain compare two keywords by serp overlap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 217.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 219 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4 else '-> different')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `serp = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compare two keywords by serp overlap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Jaccard overlap of the two keywords' ranking domains estimates shared intent, so a high overlap argues for one page covering both keywords rather than two competing ones.

```
serp = {
    'seo tools': ['ahrefs.com', 'semrush.com', 'moz.com'],
    'best seo tools': ['ahrefs.com', 'semrush.com', 'forbes.com'],
    'keyword research': ['backlinko.com', 'ahrefs.com', 'moz.com'],
}
a, b = 'seo tools', 'best seo tools'
overlap = set(serp[a]) & set(serp[b])
sim = len(overlap) / len(set(serp[a]) | set(serp[b]))
print(f'{a!r} vs {b!r}')
print('shared domains:', sorted(overlap))
print('serp similarity:', round(sim, 2), '-> same intent' if sim >= 0.4 else '-
> different')
```

```
output
'seo tools' vs 'best seo tools'
shared domains: ['ahrefs.com', 'semrush.com']
serp similarity: 0.5 -> same intent
```

How it works:

1. The serp dict maps each keyword to its top-ranking domains.
2. a and b name two keywords to compare, and overlap is the intersection of their domain sets.
3. sim divides the overlap size by the union size of both domain sets.
4. The code prints the shared domains and the rounded similarity with an inline same-or-different intent verdict.

Try it:

1. Lower the same-intent threshold and recheck the verdict.
2. Compare all keyword pairs and report the most similar.
3. Weight overlap by ranking position so top spots count more.
4. Compare 'seo tools' with 'keyword research' and watch the similarity drop.

⚠ Common Mistakes

- Dividing by `len(a) + len(b)` instead of the union size overstates the denominator.
- Comparing the ranked lists as ordered (position-sensitive) when only domain presence matters here.
- Picking the 0.4 threshold arbitrarily; real tools tune it against labelled SERP data.

Skills gained SERP overlap | set similarity | intent comparison

→ Where You'll Use This: SERP-overlap analysis decides whether near-synonym keywords can share one page or need separate ones, a core call in content consolidation work.

Mental Model

Like noticing two job ads draw the same shortlist of applicants: if Google serves the same pages, the queries are after the same thing, so merge the effort.

Project 219 Classify Keywords by Word Count into Head, Body, Tail

The Idea Behind It

Keyword length predicts behaviour: one-word head terms are high-volume and hard, while long tails are specific and easy. The loop counts words and assigns a type of head, body, or long-tail via cascading checks at one and three words. An order dict then sorts the keywords by type rank, so the printout structures the whole list from broad head terms down to specific long tails.

What You'll Build & Learn

Split keywords into head, body, and long-tail by word count and sort by that type.

- Length predicts keyword behaviour
- Three-way banding by word count
- Sorting by a custom type order

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 218: compare two keywords by serp overlap. Project 220 changes the surface problem while preserving the underlying method.

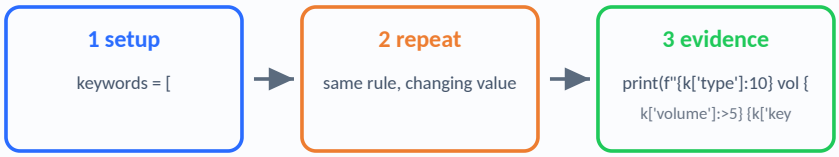
progress check

Reading target: classify keywords by word count into head, body, tail is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f"{k['type']:10} vol {k['volume']:>5} {k['keyword']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if n <= 1: k['type'] = 'head'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{k['type']:10} vol {k['volume']:>5} {k['keyword']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if n <= 1: k['type'] = 'head'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Classify Keywords by Word Count into Head, Body, Tail: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, seo, volume, tips. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = [| Core: if n <= 1: k['type'] = 'head' | Evidence: print(f"{k['type']:10} vol {k['volume']:>5} {k['keyword']}")

🛠️ Reader to Writer Bridge

1. Read: explain classify keywords by word count into head, body, tail in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 218.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 220 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{k['type']:10} vol {k['volume']:>5} {k['keyword']}")`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify keywords by word count into head, body, tail and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Word count drives a three-band label, and a small order map lets you sort by that label from head down to long-tail, mirroring the demand curve.

```
keywords = [
    {'keyword': 'seo', 'volume': 50000},
    {'keyword': 'seo tips', 'volume': 8000},
    {'keyword': 'seo tips for beginners', 'volume': 1200},
    {'keyword': 'free seo tips for small business', 'volume': 200},
]
for k in keywords:
    n = len(k['keyword'].split())
    if n <= 1: k['type'] = 'head'
    elif n <= 3: k['type'] = 'body'
    else: k['type'] = 'long-tail'
order = {'head': 0, 'body': 1, 'long-tail': 2}
for k in sorted(keywords, key=lambda k: order[k['type']]):
    print(f"{k['type']:10} vol {k['volume']:>5}  {k['keyword']}")
```

```
output
head       vol 50000  seo
body       vol  8000  seo tips
long-tail  vol  1200  seo tips for beginners
long-tail  vol   200  free seo tips for small business
```

How it works:

1. A list of dicts holds keyword and volume.
2. Each keyword's word count sets type to head at one word, body up to three, else long-tail.
3. An order dict maps each type to a sort rank.
4. sorted by that rank prints type, volume, and keyword from head down to long-tail.

Try it:

1. Adjust the band boundaries and observe how terms reclassify.
2. Sum volume per type to see where demand concentrates.
3. Add a fourth band for keywords beyond six words.
4. Shift the body/tail boundary to 4 words and re-run.

⚠ Common Mistakes

- Boundary off-by-one: a two-word keyword is 'body' here because head is `<= 1`.
- Sorting by the label string alphabetically would scramble the head-body-tail order.
- Counting characters instead of words mislabels one long word as long-tail.

Skills gained keyword segmentation | word-count banding | custom sort order

→ Where You'll Use This: Head, body, and tail segmentation shapes content strategy, balancing competitive head bets against the easier long-tail wins that build early traffic.

⚡ Mental Model

Like sorting fish by size before market: the big head terms are the prize catch but hard to land, while the small fry are plentiful and quick to sell.

Project 220 Assign Priority Tiers to a Ranked Keyword List

💡 The Idea Behind It

A flat ranked list is hard to action, so this slices a ranking into priority tiers. The priorities dict pairs keywords with scores; sorting by negative score then keyword produces a stable order. enumerate gives each a rank number, and an inline conditional assigns P1 to the top two, P2 to the next two, and P3 beyond, telling a team what to tackle this sprint versus later.

What You'll Build & Learn

Assign P1, P2, and P3 tiers to a ranked keyword list based on rank position.

- Tiers make a ranking actionable
- enumerate for rank numbers
- Position-based tier assignment

🧭 Skill Roadmap

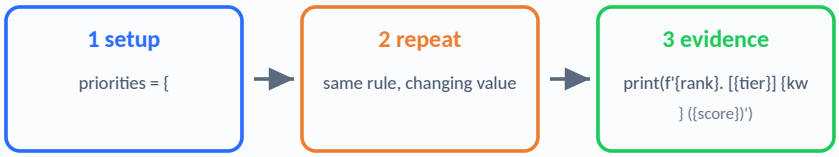
Skill step: pushes the same idea into a less automatic situation. It builds on Project 219: classify keywords by word count into head, body, tail. Project 221 changes the surface problem while preserving the underlying method. *progress check*

Reading target: assign priority tiers to a ranked keyword list is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `priorities = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{rank}. [{tier}] {kw} ({score})')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`priorities = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`tier = 'P1' if rank <= 2 else 'P2' if rank <= 4 else 'P3'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{rank}. [{tier}] {kw} ({score})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `tier = 'P1' if rank <= 2 else 'P2' if rank <= 4 else 'P3'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Assign Priority Tiers to a Ranked Keyword List: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: priorities, seo, audit, keyword, tool. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: priorities = { | Core: tier = 'P1' if rank <= 2 else 'P2' if rank <= 4 else 'P3' | Evidence: print(f'{rank}. [{tier}] {kw} ({score})')

🛠️ Reader to Writer Bridge

1. Read: explain assign priority tiers to a ranked keyword list in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 219.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 221 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{rank}. [{tier}] {kw} ({score})')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `priorities = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of assign priority tiers to a ranked keyword list and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Sorting by descending score (with the keyword as a tiebreaker) and slicing rank positions into tiers turns a continuous score into a clear work queue.

```
priorities = {
    'seo audit': 88.5,
    'keyword tool': 72.0,
    'link building': 65.3,
    'meta tags': 91.2,
    'serp tracking': 54.0,
}
ranked = sorted(priorities.items(), key=lambda x: (-x[1], x[0]))
for rank, (kw, score) in enumerate(ranked, 1):
    tier = 'P1' if rank <= 2 else 'P2' if rank <= 4 else 'P3'
    print(f'{rank}. [{tier}] {kw} ({score})')
```

```
output
1. [P1] meta tags (91.2)
2. [P1] seo audit (88.5)
3. [P2] keyword tool (72.0)
4. [P2] link building (65.3)
5. [P3] serp tracking (54.0)
```

How it works:

1. The priorities dict pairs five keywords with priority scores.
2. ranked sorts the items by negative score then keyword for a stable order.
3. enumerate starting at 1 numbers each keyword's rank.
4. An inline conditional sets P1 for ranks one to two, P2 for three to four, else P3, printed per line.

Try it:

1. Tier by score thresholds instead of by rank position.
2. Make the tier cutoffs configurable parameters.
3. Add a P0 tier for any score above ninety.
4. Change the tier cutoffs to top-1 / next-2 / rest.

⚠ Common Mistakes

- Tiering by score thresholds instead of rank position gives uneven tier sizes.

- Dropping the keyword tiebreaker makes equal scores rank unpredictably.
- Starting `enumerate` at 0 throws off the rank numbers and the tier cutoffs.

Skills gained `tier assignment` `enumerate` `ranked sort`

→ **Where You'll Use This:** Priority tiers translate a long keyword ranking into sprint planning, telling teams plainly what to ship now, next, and someday.

Mental Model

Like a triage nurse tagging patients red, yellow, and green: the ordering already exists, but the colour band tells the team who gets attention first.

Project 221 Weight Volume by Intent to Get Effective Demand

The Idea Behind It

Raw volume overrates low-intent traffic, so multiplying each keyword's volume by an intent weight gives effective demand. A weights dict scores each intent class, a separate vols dict holds the volumes, and the loop multiplies the looked-up weight by the volume per keyword. A navigational term with 6000 volume drops below a 4000-volume informational query once intent is weighed in.

What You'll Build & Learn

Compute effective demand by weighting each keyword's volume by its intent class.

- Effective demand discounts low intent
- Weighting volume by intent class
- Joining two dicts by key

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 220: assign priority tiers to a ranked keyword list. Project 222 changes the surface problem while preserving the underlying method.

progress check

Reading target: weight volume by intent to get effective demand is the surface task; the transferable Python move is to mark setup, rule, and output before running.

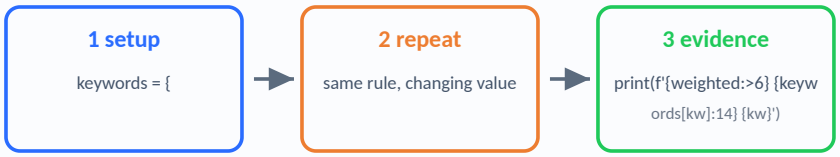
setup -> rule -> result

Logic Lens

- Setup: `keywords = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{weighted:>6} {keywords[kw]:14} {kw}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for kw in sorted(keywords):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{weighted:>6} {keywords[kw]:14} {kw}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for kw in sorted(keywords):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Weight Volume by Intent to Get Effective Demand: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, how, to, start, blog. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = { | Core: for kw in sorted(keywords): | Evidence: print(f'{weighted:>6} {keywords[kw]:14} {kw}')

🛠️ Reader to Writer Bridge

1. Read: explain weight volume by intent to get effective demand in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 220.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 222 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{weighted:>6} {keywords[kw]:14} {kw}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `keywords = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of weight volume by intent to get effective demand and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Looking up each keyword's intent weight and multiplying by its volume produces effective demand, where a smaller transactional term can outweigh a larger informational one.

```
keywords = {
    'how to start a blog': 'informational',
    'best blogging platform': 'commercial',
    'wordpress pricing': 'transactional',
    'wix login': 'navigational',
}
weights = {'informational': 0.5, 'commercial': 1.5, 'transactional': 2.0,
'navigational': 0.3}
vols = {'how to start a blog': 9000, 'best blogging platform': 4000, 'wordpress
pricing': 2000, 'wix login': 6000}
for kw in sorted(keywords):
    weighted = round(vols[kw] * weights[keywords[kw]])
    print(f'{weighted:>6}  {keywords[kw]:14} {kw}')
```

```
output
  6000  commercial     best blogging platform
  4500  informational  how to start a blog
  1800  navigational   wix login
  4000  transactional  wordpress pricing
```

How it works:

1. The keywords dict maps each keyword to its intent label.
2. weights scores each intent class and vols holds each keyword's volume.
3. For each sorted keyword, weighted is its volume times its intent's weight, rounded.
4. Each line prints the weighted demand, the intent, and the keyword.

Try it:

1. Tune the weights to match your conversion data.
2. Sort the output by weighted demand rather than by keyword.
3. Add an unknown-intent class with a neutral weight of one.
4. Sort the output by weighted demand descending.

⚠ Common Mistakes

- An intent value missing from the weights dict raises KeyError.
- Weighting all intents equally collapses effective demand back to raw volume.
- Iterating without sorting the keys gives nondeterministic line order.

Skills gained `effective demand` `intent weighting` `dict join`

→ **Where You'll Use This:** Intent-weighted volume reranks a keyword list by likely business impact, demoting high-traffic but low-value navigational and informational terms.

⚡ Mental Model

Like adjusting nominal salaries for cost of living: the headline number flatters until you weight it by what it's really worth in context.

Project 222 Find Reachable Keywords by Difficulty and Current Rank

💡 The Idea Behind It

A keyword is realistically reachable if it is not too hard and you are either unranked or already close. A reachable function returns true when difficulty is at most 45 and rank is either 0 (unranked) or within 20. A comprehension keeps the reachable keywords, sorts them by negative volume, and prints them, while the skipped terms are listed separately, building an honest target list.

What You'll Build & Learn

Build an honest target list of keywords that pass both a difficulty cap and a rank window.

- Reachability combines difficulty and rank
- Treating unranked as a special case
- Separating kept from skipped keywords

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 221: weight volume by intent to get effective demand. Project 223 changes the surface problem while preserving the underlying method. *progress check*

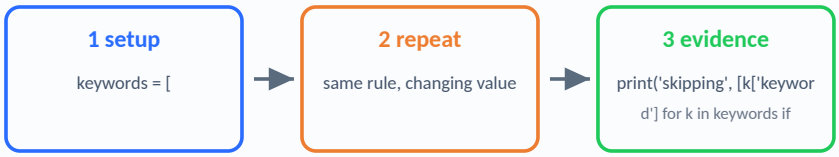
Reading target: find reachable keywords by difficulty and current rank is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print('skipping', [k['keyword'] for k in keywords if not reachable(k)])` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`wins = [k for k in keywords if reachable(k)]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('skipping', [k['keyword'] for k in keywords if not reachable(k)])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `wins = [k for k in keywords if reachable(k)]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Find Reachable Keywords by Difficulty and Current Rank: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, cheap, flights, volume. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: keywords = [| Core: wins = [k for k in keywords if reachable(k)] | Evidence: print('skipping', [k['keyword'] for k in keywords if not reachable(k)])

🛠️ Reader to Writer Bridge

1. Read: explain find reachable keywords by difficulty and current rank in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 221.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 223 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('skipping', [k['keyword'] for k in keywords if not reachable(k)])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find reachable keywords by difficulty and current rank and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The `reachable` predicate folds a difficulty ceiling and a rank window into one test, where rank 0 (unranked) is treated as a fresh opportunity rather than an existing position.

```
keywords = [
    {'keyword': 'cheap flights', 'volume': 20000, 'difficulty': 85, 'rank': 0},
    {'keyword': 'cheap flights to rome', 'volume': 3000, 'difficulty': 40,
'rank': 12},
    {'keyword': 'last minute flights rome', 'volume': 900, 'difficulty': 25,
'rank': 6},
]
def reachable(k):
    return k['difficulty'] <= 45 and (k['rank'] == 0 or k['rank'] <= 20)
wins = [k for k in keywords if reachable(k)]
for k in sorted(wins, key=lambda k: -k['volume']):
    print(f"reachable: {k['keyword']} (diff {k['difficulty']}, rank
{k['rank']})")
print('skipping', [k['keyword'] for k in keywords if not reachable(k)])
```

```
output
reachable: cheap flights to rome (diff 40, rank 12)
reachable: last minute flights rome (diff 25, rank 6)
skipping ['cheap flights']
```

How it works:

1. A list of dicts holds keyword, volume, difficulty, and rank.
2. reachable returns true when difficulty is 45 or less and rank is 0 or within 20.
3. wins keeps the reachable keywords, sorted by negative volume.
4. Each reachable keyword prints, then the skipped keywords print as a list.

Try it:

1. Treat an unranked keyword as lower priority than a close one.
2. Make the difficulty cap and rank window tunable.
3. Sort reachable keywords by current rank instead of volume.
4. Tighten the difficulty ceiling to 30 and recount the reachable set.

⚠ Common Mistakes

- Treating rank 0 as the best position (not 'unranked') inverts the rank logic.
- Using `or` between the difficulty and rank tests admits keywords that fail one condition.
- A keyword at rank exactly 20 is included because the test is `<= 20`.

Skills gained reachability filter compound condition kept-versus-skipped split

→ **Where You'll Use This:** Reachability filtering keeps a roadmap honest by dropping unwinnable head terms and surfacing keywords where effort can plausibly move the ranking.

⚡ Mental Model

Like a climber picking routes within their grade and near their current foothold: too hard or too far off, and it goes on the someday list, not today's.

Project 223 Cluster Keywords by Any Shared Token

💡 The Idea Behind It

A keyword can belong to several clusters at once, so this indexes every keyword under each of its meaningful tokens. A defaultdict of sets adds each keyword to the set for every word it contains, skipping stopwords like best, for, and the. Filtering to tokens with at least two keywords keeps the real clusters, so 'best running shoes' shows up under both running and shoes, revealing overlapping groups.

What You'll Build & Learn

Cluster keywords under every meaningful token they contain, allowing multi-cluster membership.

- Keywords can join multiple clusters
- defaultdict(set) for token indexing
- Stopword skipping and a size filter

🧭 Skill Roadmap

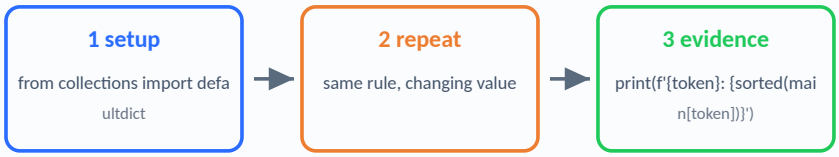
Skill step: pushes the same idea into a less automatic situation. It builds on Project 222: find reachable keywords by difficulty and current rank. Project 224 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cluster keywords by any shared token is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f'{token}: {sorted(main[token])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if token not in ('best', 'for', 'the'):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{token}: {sorted(main[token])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if token not in ('best', 'for', 'the'):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Cluster Keywords by Any Shared Token: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: collections, defaultdict, keywords, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: if token not in ('best', 'for', 'the'): | Evidence: print(f'{token}: {sorted(main[token])}')

🛠️ Reader to Writer Bridge

1. Read: explain cluster keywords by any shared token in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 222.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 224 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{token}: {sorted(main[token])}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of cluster keywords by any shared token and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Adding each keyword to a set under every non-stop-word token produces overlapping clusters; filtering to tokens with two or more members keeps only the meaningful groups.

```
from collections import defaultdict
keywords = ['running shoes', 'best running shoes', 'trail running shoes', 'kids shoes', 'kids socks', 'best socks']
clusters = defaultdict(set)
for kw in keywords:
    for token in kw.split():
        if token not in ('best', 'for', 'the'):
            clusters[token].add(kw)
main = {t: kws for t, kws in clusters.items() if len(kws) >= 2}
for token in sorted(main):
    print(f'{token}: {sorted(main[token])}')
```

```
output
kids: ['kids shoes', 'kids socks']
running: ['best running shoes', 'running shoes', 'trail running shoes']
shoes: ['best running shoes', 'kids shoes', 'running shoes', 'trail running shoes']
socks: ['best socks', 'kids socks']
```

How it works:

1. A defaultdict(set) named clusters indexes keywords by token.
2. For each keyword, every token except stopwords like best, for, and the is added under that token.
3. main keeps only tokens whose keyword set has two or more members.
4. Each surviving token prints with its sorted keywords.

Try it:

1. Expand the stopword set with more glue words.
2. Raise the minimum cluster size to three keywords.
3. Report how many clusters each keyword belongs to.
4. Drop the stop-word filter and watch 'best' become a cluster.

⚠ Common Mistakes

- Using a list instead of a set lets a repeated token add the same keyword twice.
- Not filtering stop words puts 'best' and 'for' at the top as noise clusters.
- Keeping single-keyword tokens clutters the output with trivial clusters.

Skills gained multi-token clustering | defaultdict of sets | stopword filtering

→ Where You'll Use This: Multi-token clustering builds overlapping topic maps that mirror how real content interlinks, beyond the rigid one-bucket-per-keyword view of simpler methods.

Mental Model

Like tagging a photo with several labels at once: a beach sunset lives under both 'beach' and 'sunset', so it surfaces in more than one album.

Project 224 Forecast Traffic Uplift from Rank Improvements

The Idea Behind It

The point of ranking higher is more clicks, so this quantifies the uplift each optimisation delivers. Using a position-to-CTR table, the code estimates current traffic from the current rank and projected traffic from the target rank, with dict.get supplying a tiny fallback CTR for unlisted positions. The difference is the gain per keyword, summed into a total uplift that justifies the work.

What You'll Build & Learn

Forecast the traffic uplift from moving keywords from their current to a target rank.

- Uplift is target traffic minus current
- CTR-table lookup with a fallback
- Summing per-keyword gains

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 223: cluster keywords by any shared token. Project 225 changes the surface problem while preserving the underlying method.

progress check

Reading target: forecast traffic uplift from rank improvements is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print('total uplift:', sum(k['gain'] for k in keywords))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`for k in keywords:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('total uplift:', sum(k['gain'] for k in keywords))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for k in keywords:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Forecast Traffic Uplift from Rank Improvements: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: ctr, keywords, keyword, seo, guide. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05} | Core: for k in keywords: | Evidence: print('total uplift:', sum(k['gain'] for k in keywords))

🛠️ Reader to Writer Bridge

1. Read: explain forecast traffic uplift from rank improvements in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 223.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 225 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('total uplift:', sum(k['gain'] for k in keywords))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of forecast traffic uplift from rank improvements and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Estimating clicks at both positions via the CTR table (with a tiny default for off-table ranks) and subtracting gives the per-keyword uplift, summed into a total business case.

```
ctr = {1: 0.28, 2: 0.15, 3: 0.10, 4: 0.07, 5: 0.05}
keywords = [
    {'keyword': 'seo guide', 'volume': 12000, 'current': 8, 'target': 3},
    {'keyword': 'keyword tool', 'volume': 6000, 'current': 5, 'target': 2},
]
for k in keywords:
    now = round(k['volume'] * ctr.get(k['current'], 0.01))
    later = round(k['volume'] * ctr[k['target']])
    k['gain'] = later - now
    print(f"{k['keyword']}: {now} -> {later} traffic (+{k['gain']})")
print('total uplift:', sum(k['gain'] for k in keywords))
```

```
output
seo guide: 120 -> 1200 traffic (+1080)
keyword tool: 300 -> 900 traffic (+600)
total uplift: 1680
```

How it works:

1. The ctr dict maps positions to click-through rates, and keywords hold volume, current, and target ranks.
2. now uses ctr.get on the current rank with a small fallback; later uses the target rank's CTR.
3. k['gain'] is later minus now, printed per keyword.
4. The total uplift sums every keyword's gain.

Try it:

1. Apply a confidence discount to the projected gains.
2. Forecast revenue uplift by adding conversion and value fields.
3. Sort keywords by their gain to prioritise the work.
4. Extend the CTR table to position 8 with a real value.

⚠ Common Mistakes

- Looking up `ctr[k['current']]` without `.get` raises KeyError for an off-table rank like 8.

- Using the same position for now and later yields a zero, hiding the uplift.
- Forgetting `round` leaves fractional clicks in the forecast.

Skills gained `uplift forecasting` `CTR lookup` `gain summation`

→ **Where You'll Use This:** Uplift forecasts build the business case for SEO work, translating planned ranking gains into a concrete, defensible estimate of extra monthly clicks.

Mental Model

Like estimating the extra customers from moving your shop from a side street to the high street: same store, better position, and the footfall math proves the move.

Project 225 Build a Full Keyword Prioritization Report

The Idea Behind It

This capstone ties the chapter together. An intent function classifies each keyword from its trigger words, a weights dict scores those intents, and the loop computes score as volume times intent weight divided by difficulty. Sorting by negative score then keyword prints a single ranked decision table with a header, so classification, weighting, and difficulty fuse into the one prioritised view a keyword strategy delivers.

What You'll Build & Learn

Produce a full prioritisation report combining intent classification, intent weighting, and difficulty.

- Composing intent, weight, and difficulty
- A reusable intent classifier
- Printing a headed decision table

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 224: forecast traffic uplift from rank improvements. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

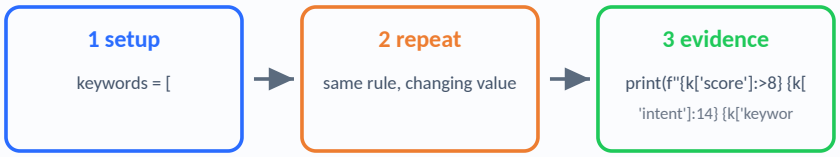
Reading target: build a full keyword prioritization report is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `keywords = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
- Evidence: `print(f"{k['score']:>8} {k['intent']:14} {k['keyword']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`keywords = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
2. Apply the rule	`if w & {'buy', 'price', 'order'}: return 'transactional'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{k['score']:>8} {k['intent']:14} {k['keyword']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Keyword Research and Intent as Data becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if w & {'buy', 'price', 'order'}: return 'transactional'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Build a Full Keyword Prioritization Report: connect the surface task to the chapter skill, score, prioritise and cluster keywords; classify serp intent; model difficulty against volume.
Names	Watch these names first: keywords, keyword, best, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: keywords = [| Core: if w & {'buy', 'price', 'order'}: return 'transactional' | Evidence: print(f" {k['score']:>8} {k['intent']:14} {k['keyword']}")

🛠️ Reader to Writer Bridge

1. Read: explain build a full keyword prioritization report in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 224.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{k['score']:>8} {k['intent']:14} {k['keyword']}")`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `keywords = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a full keyword prioritization report and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each keyword is classified, weighted, and scored in one pass, then a single sort by descending score (with a keyword tiebreaker) produces the ranked decision table that drives the content plan.

```
keywords = [
    {'keyword': 'best running shoes', 'volume': 8000, 'difficulty': 45},
    {'keyword': 'buy running shoes online', 'volume': 3000, 'difficulty': 30},
    {'keyword': 'what are running shoes', 'volume': 2000, 'difficulty': 15},
    {'keyword': 'nike store', 'volume': 12000, 'difficulty': 70},
]
weights = {'informational': 0.5, 'commercial': 1.5, 'transactional': 2.0,
'navigational': 0.3}
def intent(kw):
    w = set(kw.split())
    if w & {'buy', 'price', 'order'}: return 'transactional'
    if w & {'best', 'top', 'review'}: return 'commercial'
    if w & {'what', 'how', 'why'}: return 'informational'
    return 'navigational'
for k in keywords:
    k['intent'] = intent(k['keyword'])
    k['score'] = round(k['volume'] * weights[k['intent']] / k['difficulty'], 1)
print(f"{'SCORE':>8}  {'INTENT':14} KEYWORD")
for k in sorted(keywords, key=lambda k: (-k['score'], k['keyword'])):
    print(f"{k['score']:>8}  {k['intent']:14} {k['keyword']}")
```

```
output
   SCORE  INTENT         KEYWORD
   266.7  commercial     best running shoes
   200.0  transactional  buy running shoes online
    66.7  informational  what are running shoes
    51.4  navigational   nike store
```

How it works:

1. A list of dicts holds keyword, volume, and difficulty, with a weights dict per intent.
2. intent classifies each keyword from its word set into one of four classes.
3. The loop writes intent and a score of volume times that intent's weight over difficulty into each dict.
4. A header prints, then keywords sorted by negative score and keyword print as the ranked table.

Try it:

1. Add a CPC factor so commercial value shapes the score.
2. Append a priority tier column to the table.
3. Export the ranked rows to a CSV file.
4. Add a CPC field and fold it into the score for a value-weighted ranking.

⚠ Common Mistakes

- A 12000-volume navigational term ('nike store') scores low because of its 0.3 weight and high difficulty, which surprises volume-only thinkers.
- Forgetting the keyword tiebreaker would make any two equal scores order nondeterministically.
- An intent that has no entry in the weights dict raises KeyError mid-report.

Skills gained prioritisation report | intent weighting | decision table

→ Where You'll Use This: This composite report is the deliverable a keyword research engagement ends on, a ranked table fusing intent and difficulty that a team can act on directly.

Mental Model

Like a final scorecard at the end of a competition: every earlier judging round folds into one ranked sheet that names the winner without further argument.

✅ Chapter Progress Checkpoint

What this chapter added: after 35 mini-projects, Keyword Research and Intent as Data should feel like a reusable Python move: score, prioritise and cluster keywords; classify serp intent; model difficulty against volume. *chapter checkpoint*

1 Read
Open any project from Projects 191-225 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.
transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 7 (Canonicalization, Duplicate and Thin Content) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

6 questions · 4 categories · graded

2 Recall | 2 Meaning | 1 Structure | 1 Transfer

6.1.Q1 Recall MEDIUM
Turn back to Project 204 (Split Branded from Non-Branded Keywords). Without running it, what exactly does it print?

6.1.Q2 Meaning MEDIUM
Turn back to Project 198 (Measure Keyword-Set Overlap with Jaccard). Without running it, what exactly does it print?

6.1.Q3 Structure EASY Someone searches "buy nike air max". What is the search intent, and what page should rank?

- A Informational — a blog guide
- B Navigational — the homepage
- C Transactional — a product/checkout page
- D Commercial — a comparison post

6.1.Q4 Transfer HARD Keyword A: volume 4000, difficulty 30, intent_weight 3. Keyword B: volume 9000, difficulty 60, intent_weight 2. Using score = volume × weight ÷ difficulty, which ranks higher?

- A A (400) over B (300)
- B B (300) over A (400)
- C They tie
- D B, because it has more volume

6.1.Q5 Recall EASY Why are long-tail keywords (4+ words) often worth targeting despite low volume?

- A They are illegal to skip
- B They have lower difficulty and sharper intent, so they are easier to rank for and convert better
- C They always have higher volume
- D Search engines ignore short keywords

6.1.Q6 Meaning MEDIUM You find that /running-shoes and /shoes/running both target "best running shoes". What problem is this and how do you fix it?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 6 – Keyword Research and Intent as Data

6.1.Q1. `branded (3): ['acme login', 'acme reviews', 'acme running shoes']` then `non-branded (2): ['best running shoes', 'cheap trainers']`

6.1.Q2. `shared: ['best shoes', 'running shoes']` then `jaccard: 0.4`

6.1.Q3. C "buy" is a transactional trigger; the searcher is ready to act, so a product page (not a guide) should rank.

6.1.Q4. A A scores 4000×3÷30 = 400 and B scores 9000×2÷60 = 300, so A ranks higher despite lower volume.

6.1.Q5. B long-tail terms face less competition and the searcher is more specific, so they rank more easily and convert at a higher rate.

6.1.Q6. Keyword cannibalization — two of your own pages compete and split rankings. Fix by choosing one canonical page and merging or 301-redirecting the other into it.

Chapter 7 · Canonicalization, Duplicate and Thin Content

Check canonical consistency, detect duplicates with shingling/Jaccard, and flag thin pages.

Chapter Skill Map

What this chapter adds: In Canonicalization, Duplicate and Thin Content, the focus is check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages. You practise it across 38 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 7 assumes the reading habits from Chapter 6: Keyword Research and Intent as Data. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 8, Site Speed and Core Web Vitals, reuses this chapter's discipline with a new kind of problem.

1 **Projects 226-237**	2 **Projects 238-249**	3 **Projects 250-263**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

One page, many URLs is the default state of the web — your job is to collapse the copies and keep the credit on one URL. *dedupe, then consolidate*

The same page hides behind a dozen addresses. `https://Example.com/Page/`, `https://example.com/page`, the `www.` version, the `?utm_source=ad` version, the trailing-slash version — every one of them serves the *same* content, yet a naive crawler sees a dozen distinct URLs. That is duplication, and it costs you twice: it burns crawl budget fetching the same bytes over and over, and it splits ranking signals — links and relevance that should pile onto one URL scatter across the copies, so none of them ranks as well as the consolidated page would.

The cure has two halves. First, URL normalization: a deterministic function that rewrites every variant into one canonical string, so duplicates become *literally equal* and a dictionary can group them. Second, the canonical tag (`<link rel="canonical">`): a hint in each page's HTML naming which URL is the representative, so the engine folds the duplicate's signals onto the chosen one. A healthy ordinary page points its canonical at *itself* — a self-referencing canonical is the baseline of a clean site.

Normalization is just string surgery, applied in a fixed order. Lowercase the host (hostnames are case-insensitive), drop the fragment, sort the query parameters, and trim the trailing slash. Do it the same way every time and the variants snap together:

```
url = 'https://Example.COM/Blog/Post/?b=2&a=1#section'

def normalize(u):
```

```
    scheme, rest = u.split('://', 1)
    rest = rest.split('#', 1)[0]            # drop the fragment
    if '?' in rest:
        hostpath, query = rest.split('?', 1)
        query = '?' + '&'.join(sorted(query.split('&')))
    else:
        hostpath, query = rest, ''
    host, slash, path = hostpath.partition('/')
    path = '/' + path
    if len(path) > 1 and path.endswith('/'):
        path = path[:-1]                    # trim non-root trailing slash
    return scheme + '://' + host.lower() + path + query

print(normalize(url))
```

```
output
https://example.com/Blog/Post?a=1&b=2
```

Notice the host is lowercased but the *path* is not — `/Blog/Post` stays cased because paths are case-sensitive on most servers. Get the order and the rules right and exact duplicates collapse to one key.

normalize the host, not the path

Exact copies are the easy case. The harder problem is the near-duplicate — pages that differ by a word or two ("for every home" vs "for every house"), which a hash will not catch because one changed character flips the whole digest. For those you need a *similarity* measure: break each page into overlapping word windows called shingles, then compute the Jaccard similarity — shared shingles over total shingles, on a 0-to-1 scale. Above a threshold, the pages are near-duplicates of each other. And running underneath both is thin content: pages with so few unique words they satisfy no searcher, which an engine treats as low-value and may decline to index.

CANONICAL TAG

A `<link rel="canonical" href="...">` in a page's HTML that names which URL is the indexing representative. The engine folds duplicate signals onto that target. An ordinary page should self-reference (canonical == its own URL).

URL NORMALIZATION

A deterministic function that rewrites every variant of a URL (case, slash, query order, fragment, tracking params, www, default port) into one canonical string, so duplicates become literally equal.

DUPLICATE CONTENT

Two or more URLs serving byte-identical (after normalising case/whitespace) visible text. Detected with a content hash: identical text → identical digest.

NEAR-DUPLICATE

Pages whose text differs only slightly. A hash misses these, so similarity is measured with shingles + Jaccard, MinHash or SimHash above a chosen threshold.

SHINGLE

A sliding window of k consecutive tokens (words or characters) from a page. The set of a page's shingles captures local word order, the unit every near-duplicate detector compares.

THIN CONTENT

A page with too little unique value — very few words, or mostly boilerplate nav/footer text — that rarely satisfies a searcher and looks low-value to an engine. A soft 404 is the extreme: a 200 status with no real content.

A self-referencing canonical is the baseline

Every ordinary page should carry a canonical tag that points at *its own* URL. That tells the engine "this is the representative — index me." The simplest audit is one comparison: does the canonical equal the page's URL? When it points elsewhere, the page is deliberately folding itself into another, which is right for a duplicate but a bug on a page that should stand alone.

```
pages = {
    'https://example.com/a': 'https://example.com/a',
    'https://example.com/b': 'https://example.com/canonical-b',
    'https://example.com/c': 'https://example.com/c',
}
for url, canon in pages.items():
    flag = 'SELF' if url == canon else 'OTHER'
    print(f'{flag} {url} -> {canon}')
```

```
output
SELF https://example.com/a -> https://example.com/a
OTHER https://example.com/b -> https://example.com/canonical-b
SELF https://example.com/c -> https://example.com/c
```

The canonical is only a *hint*, and it must not contradict the page's other signals. A page that redirects elsewhere yet still self-canonicalises sends two opposite messages ("go away" and "index me"); a canonical pointing at a `noindex` page, a `404`, a redirect, or a different domain is wasted or harmful. The rule of thumb: the canonical target should be a same-site URL that returns `200` and is itself indexable.

Normalize the URL, then group

Before you can declare two URLs duplicates you must agree on what "same" means. Each normalization rule kills one class of accidental variant. The biggest single source on a real site is tracking parameters — every ad campaign appends `utm_source`, `utm_medium` and friends, spawning a fresh URL for one page. Strip them before anything else:

```
url = 'https://example.com/sale?utm_source=news&id=42&utm_medium=email&page=2'
base, _, query = url.partition('?')
kept = [kv for kv in query.split('&') if not kv.startswith('utm_')]
clean = base + ('?' + '&'.join(kept) if kept else '')
print('clean:', clean)
print('dropped:', len(query.split('&')) - len(kept))
```

```
output
clean: https://example.com/sale?id=42&page=2
dropped: 2
```

The other classic folds are the same idea: `www.` vs the bare host, `http` vs `https`, the explicit default port `:443`, a mixed-case host, and query parameters in a different order (`?a=1&b=2` is the same page as `?b=2&a=1`). A full canonicaliser applies every rule at once, and *then* duplicates collapse into one dictionary key:

```
urls = [
    'https://www.Example.com/Page/',
    'https://example.com/page',
    'https://example.com/page?utm_source=x',
    'https://example.com/other',
]
def canon(u):
    scheme, rest = u.split('://', 1)
    rest = rest.split('#', 1)[0]
    hostpath, _, query = rest.partition('?')
    host, slash, path = hostpath.partition('/')
    host = host.lower()
    if host.startswith('www.'):
        host = host[4:]
    path = '/' + path
    if len(path) > 1 and path.endswith('/'):
        path = path[:-1]
    kept = [kv for kv in query.split('&') if kv and not kv.startswith('utm_')]
    q = '?' + '&'.join(sorted(kept)) if kept else ''
    return scheme + '://' + host + path.lower() + q
```

```
clusters = {}
for u in urls:
    clusters.setdefault(canon(u), []).append(u)
for c in sorted(clusters):
    print(f'{c} <- {sorted(clusters[c])}')
```

```
output
https://example.com/other <- ['https://example.com/other']
https://example.com/page <- ['https://example.com/page', 'https://example.com/page?
utm_source=x', 'https://www.Example.com/Page/']
```

Exact duplicates: hash the content

Once URLs are normalized, compare the *content*. Two pages with byte-identical visible text are exact duplicates. Comparing every page to every other is wasteful; instead, `fingerprint` each page — normalise whitespace and case, then take a `SHA-1` digest — and group by digest. Identical text yields an identical hash, so duplicate detection becomes a single dictionary lookup. (Use `hashlib.sha1`, never Python's builtin `hash`, which is salted per process and so is not stable across runs.)

```
import hashlib

def fingerprint(text):
    return hashlib.sha1(' '.join(text.lower().split()).encode()).hexdigest()

pages = {
    'https://example.com/a': 'Buy   Red Shoes Today',
    'https://example.com/b': 'buy red shoes today',
    'https://example.com/c': 'Buy Blue Shoes Today',
}
groups = {}
for url, text in pages.items():
    groups.setdefault(fingerprint(text), []).append(url)
for h, urls in groups.items():
    tag = 'DUP' if len(urls) > 1 else 'unique'
    print(f'{tag} {h[:8]} {sorted(urls)}')
```

```
output
DUP 42f8e1ec ['https://example.com/a', 'https://example.com/b']
unique 8f815bf4 ['https://example.com/c']
```

The normalisation step is what makes the fingerprint *stable across cosmetic edits but reactive to real ones*. "Buy Red Shoes Today" and "buy red shoes today" hash the same (good — they are the same page); adding a "!" would change the digest (also good — that is a real edit). *normalise, then hash*

Near-duplicates: shingles and Jaccard

A hash is all-or-nothing: change one word and the digest is completely different, so hashing can never catch a page that was barely tweaked. For that you measure *overlap*. Break each page into k-shingles — every window of k consecutive words — then compare the two sets. The Jaccard similarity is the size of the intersection over the size of the union: how much the two pages share, from 0 (nothing) to 1 (identical).

```
def shingles(text, k=2):
    w = text.lower().split()
    return {tuple(w[i:i + k]) for i in range(len(w) - k + 1)}

a = shingles('red running shoes for sale')
b = shingles('red running shoes on sale')
inter = a & b
union = a | b
print('shared shingles:', len(inter))
print('total shingles:', len(union))
print('jaccard:', round(len(inter) / len(union), 3))
```

```
output
shared shingles: 2
total shingles: 6
jaccard: 0.333
```

Pick a threshold (say 0.5) and any pair clearing it is a near-duplicate. Comparing every pair across a set surfaces clusters of barely-changed pages — the "we made fifty city landing pages by find-and-replacing the town name" pattern that engines penalise. For web scale you would replace the all-pairs scan with cheaper estimators (MinHash, SimHash) that approximate Jaccard from a tiny fingerprint, but the metric being approximated is always this one.

Thin content: count what is actually there

Duplication is one failure mode; thinness is the other. A page with a handful of words rarely answers a query and looks low-value to an engine. The crudest gate is a word count, and a sharper one is the boilerplate-to-content ratio — the share of words that are templated nav/footer junk ("home", "login", "privacy") rather than real content. When boilerplate dominates, the unique content is too thin to rank, even if the raw count looks fine.

```
import re
pages = {
    'https://example.com/full': 'Our complete guide explains every step of
installing solar panels safely at home this season.',
    'https://example.com/thin': 'Solar panels. Buy now.',
    'https://example.com/ok': 'Solar panels convert sunlight into electricity using
photovoltaic cells mounted on your roof.',
}
threshold = 10
for url in sorted(pages):
```

```
    wc = len(re.findall(r"[a-zA-Z']+", pages[url]))
    tag = 'THIN' if wc < threshold else 'ok'
    print(f'{tag} {wc} words {url}')
```

```
output
ok 15 words https://example.com/full
ok 13 words https://example.com/ok
THIN 4 words https://example.com/thin
```

The capstone of any audit ties all three together in one pass over the page set: confirm the canonical is self-referencing, hash the body to catch exact duplicates, and count words to flag thinness — three independent checks, one loop. *self-canonical? duplicate? thin?*

Example 1

A page at `https://example.com/b` carries a self-referencing canonical (`canonical == url`) but the server returns a 301 redirect to `https://example.com/c`. Is that page healthy?

SOLUTION

1. The canonical says "index me, I am the representative" — a normal, indexable page.
2. But the 301 redirect says the opposite: "this URL has moved, go to /c instead." The two signals `contradict`.
3. An engine cannot both index /b and follow it away to /c. The redirect should win, so /b should *not* self-canonicalise — it should drop the tag, or /c should be the canonical.

No — it is a conflict: a redirecting URL must not also self-canonicalise.

redirect and self-canonical are mutually exclusive

Example 2

Two product pages read "cheap blue widgets for every home" and "cheap blue widgets for every house". Using 2-word shingles, are they near-duplicates at a 0.5 threshold?

SOLUTION

1. Shingles of A: {(cheap,blue),(blue,widgets),(widgets,for),(for,every),(every,home)} — 5 shingles.
2. Shingles of B: the same five except the last is (every,house) instead of (every,home). The four leading shingles are `shared`.
3. Intersection = 4, union = 6, so Jaccard = 4/6 = `0.667`, which clears the 0.5 threshold.

Yes — Jaccard 0.667 ≥ 0.5, so they are flagged as near-duplicates (only the last word changed).

one word swapped, signals split.

Example 3

You collect four URLs that all serve one product page: `.../product?ref=ad&color=red`, `.../product`, `.../product/`, `.../product?color=red`. Which should you pick as the canonical, by a "shortest URL, ties broken alphabetically" policy?

SOLUTION

1. Normalise: the trailing-slash and the tracking-param variants all reduce toward the bare path, but for the *choice* you score the candidates by `(len(url), url)`.
2. The shortest is `https://example.com/product` (no query, no slash) — every other variant is strictly longer.
3. Choose it as the representative and fold the other three onto it via canonical tags.

Pick `https://example.com/product` — the shortest, cleanest URL — and canonicalise the rest to it.

one page, one chosen URL

⚡ Mental-Math Shortcut

Normalize, then consolidate. Duplicate work is wasted twice — once in crawl budget, once in split ranking signals. First make copies *literally equal* with a deterministic URL normaliser (lowercase host, drop fragment, strip `utm_`, sort params, trim slash); then *consolidate* the credit onto one representative with a canonical tag that the page's redirect, status, and indexability all agree with. Hash catches exact copies; shingles + Jaccard catch the near ones; a word count catches the thin ones.

⚠ Common Mistakes

- Using Python's builtin `hash()` for content fingerprints — it is salted per process, so the same text hashes differently across runs and your duplicate groups silently break. Use `hashlib.sha1` (or sha256).
- Lowercasing the whole URL during normalization. The host is case-insensitive, but the path usually is not, so `/Page` and `/page` can be genuinely different resources — only lowercase the host.
- Expecting a canonical tag to deduplicate by force. It is a hint, not a redirect: if the page also returns a redirect, is `noindex`, 404s, or points cross-domain, the engine may ignore or distrust it.
- Trying to catch near-duplicates with a content hash. One changed word flips the whole digest, so hashing reports "all unique" while fifty find-and-replace landing pages slip through — that is what Jaccard/shingles are for.

This chapter contains 38 hands-on mini-projects (Projects 226–263). Type each one out, run it, and experiment.

Project 226 Check a Canonical Is Self-Referencing

💡 The Idea Behind It

A healthy page declares itself the master copy by pointing its canonical tag back at its own URL. This project models one page as a dictionary holding a url and a canonical key, then compares the two strings. When they match, the page is self-referencing; when they differ, the canonical defers indexing credit to somewhere else. It is the smallest possible canonical health check and the baseline every audit starts from.

What You'll Build & Learn

Confirm a single page canonicalises to itself by comparing its url and canonical fields.

- What a self-referencing canonical means
- Comparing two URL strings for equality
- The baseline of a healthy page

Skill Roadmap

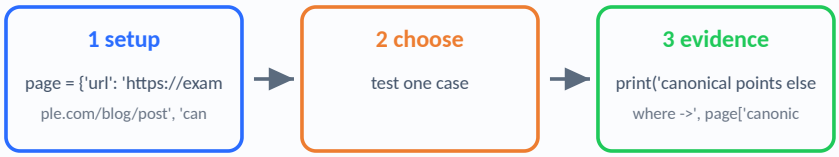
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 227 changes the surface problem while preserving the underlying method. *progress check*

Reading target: check a canonical is self-referencing is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `page = {'url': 'https://example.com/blog/post', 'canonical': 'https://example.com/blog/post'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('canonical points elsewhere ->', page['canonical'])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`page = {'url': 'https://example.com/blog/post', 'canonical': 'https://example.com/blog/post'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if page['url'] == page['canonical']:`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. slow line
3. Check the evidence	`print('canonical points elsewhere ->', page['canonical'])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if page['url'] == page['canonical']:`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Check a Canonical Is Self-Referencing: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: page, url, https, example, com. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: page = {'url': 'https://example.com/blog/post', 'canonical': 'https://e... | Core: if page['url'] == page['canonical']: | Evidence: print('canonical points elsewhere ->', page['canonical'])

🛠️ Reader to Writer Bridge

1. Read: explain check a canonical is self-referencing in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 227 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('canonical points elsewhere ->', page['canonical'])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `page = {'url': 'https://example.com/blog/post', 'canonical': 'https://example.com/blog/post'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check a canonical is self-referencing and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A self-referencing canonical means `url == canonical`; comparing the two strings is the entire check that a page is claiming itself as the indexable version.

```
page = {'url': 'https://example.com/blog/post', 'canonical':
'https://example.com/blog/post'}
if page['url'] == page['canonical']:
    print('self-referencing canonical: OK')
```

```
    else:
        print('canonical points elsewhere ->', page['canonical'])
```

```
output
self-referencing canonical: OK
```

How it works:

1. A dictionary stores the page url alongside its declared canonical URL
2. The if checks whether url equals canonical with a string comparison
3. When equal it prints an OK message confirming self-reference
4. The else branch reports where the canonical actually points instead

Try it:

1. Change the canonical to a different URL and watch the else branch fire
2. Wrap the check in a function that returns True or False
3. Add a third key for the http status and only pass pages returning 200
4. Point the canonical at `https://example.com/other` and watch the branch flip.

⚠ Common Mistakes

- Comparing only the path and ignoring the scheme or host hides a cross-origin canonical.
- Treating a missing canonical as self-referencing: an absent tag is a different state, not OK.
- Forgetting that a trailing slash or case difference makes two equal-looking URLs compare unequal.

Skills gained dict access | string equality | canonical basics

→ Where You'll Use This: Every site audit begins by confirming pages self-canonicalise; a crawler flags any page whose canonical tag silently hands authority to another URL.

Mental Model

A canonical tag is a page raising its hand to say 'I am the original.' A self-referencing one is the page pointing at itself.

Project 227 Detect a Canonical Pointing to Another URL

The Idea Behind It

Scaling the single-page check up to a whole set means walking a dictionary of url-to-canonical mappings and labelling each entry. The loop reads every pair, tags it SELF when the URL equals its canonical and OTHER when it points elsewhere, then prints an arrow line. Some OTHER entries are deliberate, folding a tracking-parameter variant into the clean URL, while others may be mistakes worth investigating.

What You'll Build & Learn

Iterate a set of pages and label each as self-referencing or deferring to another URL.

- Looping a url-to-canonical dictionary with items
- Tagging each page SELF or OTHER
- Reading intentional versus accidental canonicals

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 226: check a canonical is self-referencing. Project 228 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a canonical pointing to another url is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'{flag} {url} -> {canon}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output.

visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check

Move	Code to inspect	Why it matters
		canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`flag = 'SELF' if url == canon else 'OTHER'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{flag} {url} -> {canon}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = 'SELF' if url == canon else 'OTHER'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Canonical Pointing to Another URL: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, https, example, com, canonical. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: flag = 'SELF' if url == canon else 'OTHER' | Evidence: print(f'{flag} {url} -> {canon}')

Reader to Writer Bridge

1. Read: explain detect a canonical pointing to another url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 226.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 228 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{flag} {url} -> {canon}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a canonical pointing to another url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Mapping each URL to its declared canonical and labelling `SELF` versus `OTHER` separates pages that index themselves from those that point at a representative URL.

```
pages = {
    'https://example.com/a': 'https://example.com/a',
    'https://example.com/b': 'https://example.com/canonical-b',
    'https://example.com/c': 'https://example.com/c',
    'https://example.com/d?ref=ad': 'https://example.com/d',
}
for url, canon in pages.items():
    flag = 'SELF' if url == canon else 'OTHER'
    print(f'{flag} {url} -> {canon}')
```

```
output
SELF https://example.com/a -> https://example.com/a
OTHER https://example.com/b -> https://example.com/canonical-b
SELF https://example.com/c -> https://example.com/c
OTHER https://example.com/d?ref=ad -> https://example.com/d
```

How it works:

1. A dictionary maps four page URLs to their canonical targets
2. The for loop unpacks each url and canon pair via items

3. A ternary sets flag to SELF when url equals canon, else OTHER
4. An f-string prints the flag, source URL and arrow to the canonical

Try it:

1. Count how many pages are SELF versus OTHER and print a summary
2. Filter the output to show only the OTHER rows
3. Add a page whose query string is the only difference from its canonical
4. Sort the pages by key so the report order is alphabetical.

⚠ Common Mistakes

- Assuming `OTHER` is always wrong: a duplicate pointing to its representative is correct behaviour.
- Relying on dict order for output and forgetting it is insertion order, not sorted.
- Not stripping tracking params first, so `/d?ref=ad` looks like a distinct page from `/d`.

Skills gained dict iteration | ternary expression | f-string formatting

→ Where You'll Use This: Bulk canonical reports look exactly like this: one row per crawled URL showing whether it claims itself or points away, so an SEO can scan thousands of pages fast.

⚡ Mental Model

Like roll call in a classroom: each page either answers 'present, that's me' or points across the room to whoever should answer instead.

Project 228 Follow a Canonical Chain to Its End

💡 The Idea Behind It

When page A canonicalises to B and B to C, the signals form a chain that a crawler must resolve to the true endpoint. This project walks the chain with a while loop guarded at ten hops, appending each URL visited and stopping when a page canonicalises to itself. It prints the full path and the final canonical plus the hop count, because search engines may give up on chains that run too long.

What You'll Build & Learn

Resolve a canonical chain to its terminal URL and report how many hops it took.

- Following a chain of canonical pointers
- Guarding a loop against infinite cycles
- Why long chains are risky

Skill Roadmap

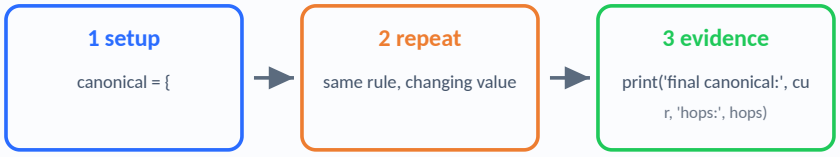
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 227: detect a canonical pointing to another url. Project 229 changes the surface problem while preserving the underlying method. *progress check*

Reading target: follow a canonical chain to its end is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `canonical = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('final canonical:', cur, 'hops:', hops)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`canonical = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if nxt == cur:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('final canonical:', cur, 'hops:', hops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nxt == cur:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Follow a Canonical Chain to Its End: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: canonical, https, example, com, p1. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: canonical = { | Core: if nxt == cur: | Evidence: print('final canonical:', cur, 'hops:', hops)

Reader to Writer Bridge

1. Read: explain follow a canonical chain to its end in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 227.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 229 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('final canonical:', cur, 'hops:', hops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `canonical = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of follow a canonical chain to its end and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Walking the canonical map until a URL canonicalises to itself resolves the true target; a hop guard stops a malformed loop from spinning forever.

```
canonical = {
    'https://example.com/p1': 'https://example.com/p2',
    'https://example.com/p2': 'https://example.com/p3',
    'https://example.com/p3': 'https://example.com/p3',
}
start, cur, chain, hops = 'https://example.com/p1', 'https://example.com/p1',
[], 0
while hops < 10:
    chain.append(cur)
    nxt = canonical.get(cur, cur)
    if nxt == cur:
        break
    cur, hops = nxt, hops + 1
print(' -> '.join(chain))
print('final canonical:', cur, 'hops:', hops)
```

```
output
https://example.com/p1 -> https://example.com/p2 -> https://example.com/p3
final canonical: https://example.com/p3 hops: 2
```

How it works:

1. A dictionary encodes a chain where p1 points to p2 and p2 to p3
2. The while loop runs while hops stays under ten as a safety cap
3. Each pass appends cur to chain then looks up the next target
4. When the next equals cur the loop breaks, marking the chain's end
5. It prints the joined path and the final URL with its hop count

Try it:

1. Insert a cycle where p3 points back to p1 and confirm the ten-hop cap stops it
2. Lower the hop cap to two and see a long chain truncate
3. Return the resolved canonical from a function for any starting URL

4. Add a fourth link `p3 -> p4` and watch the chain lengthen.

⚠ Common Mistakes

- No hop guard means a cycle (A->B->A) loops until the process is killed.
- Appending after stepping drops the start URL from the printed chain.
- Counting URLs in the chain instead of transitions overstates the hop count by one.

Skills gained `while loop` `dict.get default` `loop guard`

→ Where You'll Use This: Real audits flag canonical chains because engines often follow only a hop or two; tracing each one to its endpoint shows which pages will actually be indexed.

⚡ Mental Model

A canonical chain is a relay race of batons; the engine wants the finish line, but if the track is too long it stops watching partway.

Project 229 Flag Conflicting Canonical vs Redirect

💡 The Idea Behind It

A page that issues a redirect yet also self-canonicalises is shouting two contradictory orders: the redirect says 'leave this URL' while the canonical says 'index this URL'. The project stores each page as a nested dict with canonical and redirect fields, then flags the conflict when a redirect exists and the canonical still points at the page itself. Clean pages with no redirect, or redirects without self-canonicalisation, pass.

What You'll Build & Learn

Detect pages that redirect elsewhere while still naming themselves canonical.

- Spotting contradictory canonical and redirect signals
- Reading nested dictionary fields
- Why two signals must agree

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 228: follow a canonical chain to its end. Project 230 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag conflicting canonical vs redirect is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'OK {url}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if m['redirect'] and m['canonical'] == url:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'OK {url}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m['redirect'] and m['canonical'] == url:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Conflicting Canonical vs Redirect: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, https, example, com, canonical. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: if m['redirect'] and m['canonical'] == url: | Evidence: print(f'OK {url}')

🛠️ Reader to Writer Bridge

1. Read: explain flag conflicting canonical vs redirect in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 228.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 230 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'OK {url}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of flag conflicting canonical vs redirect and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The conflict is the conjunction of two facts: a redirect target exists AND the canonical still names the page itself; either alone is fine.

```
pages = {
    'https://example.com/a': {'canonical': 'https://example.com/a', 'redirect':
None},
    'https://example.com/b': {'canonical': 'https://example.com/b', 'redirect':
'https://example.com/c'},
    'https://example.com/d': {'canonical': 'https://example.com/d', 'redirect':
None},
}
for url, m in pages.items():
    if m['redirect'] and m['canonical'] == url:
        print(f'CONFLICT {url}: self-canonical but redirects to
{m["redirect"]}')
    else:
        print(f'OK {url}')
```

```
output
OK https://example.com/a
CONFLICT https://example.com/b: self-canonical but redirects to
https://example.com/c
OK https://example.com/d
```

How it works:

1. Each page maps to a nested dict holding its canonical and an optional redirect
2. The loop checks if a redirect is set and the canonical equals the URL
3. When both hold it prints a CONFLICT line naming the redirect target
4. Otherwise the page is reported OK

Try it:

1. Add a page that redirects and canonicalises to the redirect target instead, and treat it as OK
2. Print the redirect destination's own canonical to follow the trail
3. Count conflicts and exit non-zero when any are found
4. Change page b's canonical to its redirect target and watch the conflict clear.

⚠ Common Mistakes

- Checking only the redirect and ignoring whether the canonical agrees with it.

- Treating an empty-string redirect as truthy if you do not normalise to `None`.
- Reporting a conflict when the canonical already points at the redirect target (that is consistent).

Skills gained `nested dict` `boolean and` `conflict detection`

→ **Where You'll Use This:** Crawlers cross-check the redirect chain against the canonical tag because a page cannot honestly say 'index me' and 'go away' at once; this catches a common migration bug.

Mental Model

Like a shop with a 'we've moved' sign on the door but a fresh 'open for business' banner in the window: customers and crawlers alike get confused.

Project 230 Normalize a URL to One Canonical Form

The Idea Behind It

The same page can hide behind dozens of URL spellings that differ only by case, a trailing slash, query-parameter order or a fragment. This normaliser splits the URL apart, drops the fragment after a hash, sorts the query parameters, lowercases the host while leaving the path intact, and strips a trailing slash from non-root paths. The result is one canonical string so two cosmetic variants become literally equal.

What You'll Build & Learn

Collapse case, slash, query-order and fragment variants of a URL into one canonical form.

- Decomposing a URL into scheme, host, path and query
- Sorting query parameters for stability
- Lowercasing only the host

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 229: flag conflicting canonical vs redirect. Project 231 changes the surface problem while preserving the underlying method. *progress check*

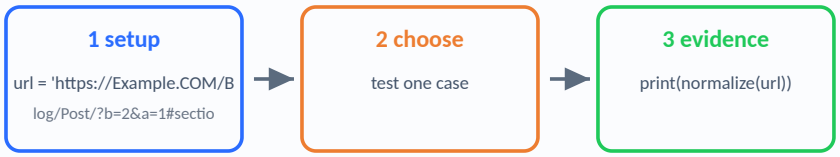
Reading target: normalize a url to one canonical form is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `url = 'https://Example.COM/Blog/Post/?b=2&a=1#section'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The function boundary is the lesson: inputs enter, a rule runs, and a return value can be reused somewhere else.

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(normalize(url))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`url = 'https://Example.COM/Blog/Post/?b=2&a=1#section'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if '?' in rest:`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(normalize(url))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if '?' in rest:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Normalize a URL to One Canonical Form: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: url, https, Example, Blog, Post. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: url = 'https://Example.COM/Blog/Post/?b=2&a=1#section' | Core: if '?' in rest: | Evidence: print(normalize(url))

🛠️ Reader to Writer Bridge

1. Read: explain normalize a url to one canonical form in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 229.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 231 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(normalize(url))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `url = 'https://Example.COM/Blog/Post/?b=2&a=1#section'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of normalize a url to one canonical form and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Dropping the fragment, sorting query params, lowercasing only the host, and trimming a non-root trailing slash produces a canonical form that is stable across cosmetic variants.

```
url = 'https://Example.COM/Blog/Post/?b=2&a=1#section'

def normalize(u):
    scheme, rest = u.split('://', 1)
    rest = rest.split('#', 1)[0]
    if '?' in rest:
        hostpath, query = rest.split('?', 1)
        params = sorted(query.split('&'))
        query = '?' + '&'.join(params)
    else:
        hostpath, query = rest, ''
    host, slash, path = hostpath.partition('/')
    path = '/' + path
    if len(path) > 1 and path.endswith('/'):
        path = path[:-1]
    return scheme + '://' + host.lower() + path + query

print(normalize(url))
```

```
output
https://example.com/Blog/Post?a=1&b=2
```

How it works:

1. The function splits scheme from the rest then drops everything after the hash fragment
2. If a question mark exists it splits and sorts the ampersand-separated params
3. partition separates host from path, and the path is reassembled with a leading slash
4. A trailing slash is trimmed from any path longer than the root
5. It returns scheme, lowercased host, path and the sorted query joined back together

Try it:

1. Also lowercase the path and compare results on a mixed-case path
2. Strip a leading www from the host as part of normalisation
3. Decode percent-encoded characters before sorting the query
4. Add a `utm_source` param and decide whether to drop it before sorting.

⚠ Common Mistakes

- Lowercasing the whole URL corrupts case-sensitive paths; only the host is safe to lowercase.
- Stripping the trailing slash from the root path `'/'` would wrongly empty it.
- Sorting query params as one string instead of splitting on `&` reorders characters, not parameters.

Skills gained `str.split` `str.partition` `param sorting`

→ **Where You'll Use This:** URL normalisation underpins every deduplication pipeline; before you can group duplicates you must reduce each address to one canonical spelling the way search engines do internally.

Mental Model

Like filing names alphabetically by 'Smith, John' regardless of how someone scrawled it: messy inputs, one tidy index entry.

Project 231 Strip Tracking Params from a URL

The Idea Behind It

Marketing campaign tags such as utm_source mint a brand-new URL for every ad placement even though all of them serve one page. This project partitions the URL at its question mark, keeps only the parameters that do not start with utm_, and rebuilds a clean address, appending the query only when something survives. It also reports how many tracking parameters were dropped, which is the duplicate count you just prevented.

What You'll Build & Learn

Remove utm_ tracking parameters from a URL and count how many were stripped.

- Splitting a URL on its query string
- Filtering parameters by prefix
- Rebuilding a clean URL conditionally

Skill Roadmap

`Skill step:` adds one small variation while the chapter pattern is still new. It builds on Project 230: normalize a url to one canonical form. Project 232 changes the surface problem while preserving the underlying method.

progress check

`Reading target:` strip tracking params from a url is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

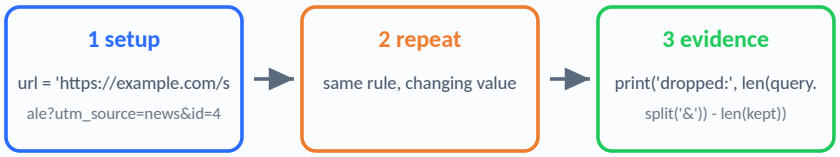
Logic Lens

- `Setup:` `url = 'https://example.com/sale?utm_source=news&id=42&utm_medium=email&page=2'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The comprehension compresses a loop into a readable noun phrase: `make this` for `each item` if `it qualifies`. Expand it mentally when it feels dense.
- `Writing move:` this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard,

and flag thin pages.

- Evidence: `print('dropped:', len(query.split('&')) - len(kept))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`url = 'https://example.com/sale?utm_source=news&id=42&utm_medium=email&page=2'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`kept = [kv for kv in query.split('&') if not kv.startswith('utm_')]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('dropped:', len(query.split('&')) - len(kept))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `kept = [kv for kv in query.split('&') if not kv.startswith('utm_')]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Strip Tracking Params from a URL: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: url, https, example, com, sale. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: url = 'https://example.com/sale?utm_source=news&id=42&utm_medium=email&... | Core: kept = [kv for kv in query.split('&') if not kv.startswith('utm_')] | Evidence: print('dropped:', len(query.split('&')) - len(kept))

🛠️ Reader to Writer Bridge

1. Read: explain strip tracking params from a url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 230.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 232 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('dropped:', len(query.split('&')) - len(kept))`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `url = 'https://example.com/sale?utm_source=news&id=42&utm_medium=email&page=2'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of strip tracking params from a url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Splitting the query on `&` and keeping only params that do not start with `utm_` removes campaign tags while preserving functional parameters like `id` and `page`.

```
url = 'https://example.com/sale?utm_source=news&id=42&utm_medium=email&page=2'
base, _, query = url.partition('?')
kept = [kv for kv in query.split('&') if not kv.startswith('utm_')]
clean = base + ('?' + '&'.join(kept) if kept else '')
print('clean:', clean)
print('dropped:', len(query.split('&')) - len(kept))
```

```
output
clean: https://example.com/sale?id=42&page=2
dropped: 2
```

How it works:

1. partition splits the URL into base, separator and the query string
2. A list comprehension keeps every key-value pair not starting with utm_
3. The clean URL re-adds a question mark and the kept params only if any remain
4. It prints the cleaned URL and the count of dropped parameters

Try it:

1. Extend the filter to also drop fbclid and gclid parameters
2. Keep a configurable allowlist of parameters that truly change the page
3. Preserve the original parameter order instead of the input order
4. Add `gclid` and `fbclid` to the strip list alongside `utm_`.

⚠ Common Mistakes

- Using `'utm_' in kv` instead of `startswith` would also drop a param like `x=utm_demo`.
- Forgetting the `'?' if kept else ''` guard leaves a dangling `?` when every param is stripped.
- Stripping all params loses pagination or product variants that genuinely change the page.

Skills gained str.partition | list comprehension | prefix filter

→ Where You'll Use This: Analytics and ad URLs flood a crawler with utm variants of one page; stripping them before indexing keeps the index lean and concentrates ranking signals on the real address.

Mental Model

Tracking tags are stickers slapped on the same product box for each store; peel them off and you see it was one box all along.

Project 232 Detect Exact Duplicates via SHA-1 of Text

The Idea Behind It

Two pages whose visible text is byte-identical after whitespace and case are flattened count as exact duplicates. This project normalises each body by lowercasing and collapsing runs of spaces, hashes the result with SHA-1, and groups URLs under their digest in a dictionary. Pages sharing a digest land in the same bucket and are tagged DUP, turning an expensive pairwise comparison into a single hash lookup.

What You'll Build & Learn

Group pages with identical normalised text by hashing their content with SHA-1.

- Normalising text before hashing
- Using a content hash as a dictionary key
- Grouping URLs by digest

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 231: strip tracking params from a url. Project 233 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect exact duplicates via sha-1 of text is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

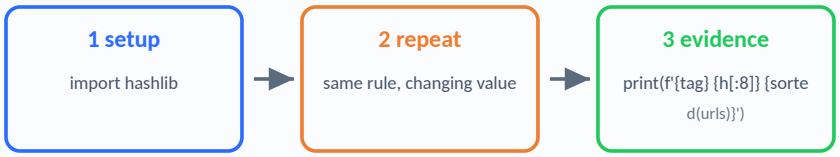
- Setup: `import hashlib` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'{tag} {h[:8]} {sorted(urls)}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import hashlib`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`tag = 'DUP' if len(urls) > 1 else 'unique'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{tag} {h[:8]} {sorted(urls)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `tag = 'DUP' if len(urls) > 1 else 'unique'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Detect Exact Duplicates via SHA-1 of Text: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: hashlib, norm, join, lower, split. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import hashlib | Core: tag = 'DUP' if len(urls) > 1 else 'unique' | Evidence: print(f'{tag} {h[:8]} {sorted(urls)}')

🛠️ Reader to Writer Bridge

1. Read: explain detect exact duplicates via sha-1 of text in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 231.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 233 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{tag} {h[:8]} {sorted(urls)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import hashlib` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect exact duplicates via sha-1 of text and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Normalising whitespace and case before `hashlib.sha1(...).hexdigest()` means cosmetically different copies hash to the same digest, so grouping by digest clusters exact duplicates.

```
import hashlib

def norm(t):
    return ' '.join(t.lower().split())
```

```
pages = {
    'https://example.com/a': 'Buy   Red Shoes Today',
    'https://example.com/b': 'buy red shoes today',
    'https://example.com/c': 'Buy Blue Shoes Today',
}
fp = {}
for url, text in pages.items():
    h = hashlib.sha1(norm(text).encode()).hexdigest()
    fp.setdefault(h, []).append(url)
for h, urls in fp.items():
    tag = 'DUP' if len(urls) > 1 else 'unique'
    print(f'{tag} {h[:8]} {sorted(urls)}')
```

```
output
DUP 42f8e1ec ['https://example.com/a', 'https://example.com/b']
unique 8f815bf4 ['https://example.com/c']
```

How it works:

1. The norm helper lowercases the text and joins split tokens to flatten whitespace
2. Each page's normalised body is encoded and hashed with hashlib.sha1
3. setdefault appends the URL to a list keyed by its hex digest
4. The final loop tags any digest shared by more than one URL as DUP

Try it:

1. Switch the digest to SHA-256 and confirm the grouping is unchanged
2. Print only the duplicate clusters and skip the unique ones
3. Strip HTML tags from each body before normalising
4. Add a fourth page identical to page c and watch its group become a DUP.

⚠ Common Mistakes

- Hashing the raw text without normalising whitespace makes `'red shoes'` and `'red shoes'` look distinct.
- Calling `.encode()` is required: `hashlib.sha1` rejects a `str` directly.
- Comparing only the first 8 hex chars for the final verdict risks a collision; the full digest groups them here.

Skills gained hashlib.sha1 | text normalization | setdefault grouping

→ Where You'll Use This: Exact-duplicate detection on a large site reduces to one hash per page; identical digests instantly reveal which URLs serve the same copy and should be canonicalised together.

Mental Model

Hashing text is like a fingerprint scanner at a door: identical fingerprints mean the same person, no matter how they dressed up the spacing.

Project 233 Build K-Shingles from Page Text

The Idea Behind It

A shingle is a sliding window of k consecutive words, and this project builds the 3-word shingles of a short sentence. It splits the text into words and uses a comprehension with a range that stops k-1 words from the end, capturing each tuple of three adjacent terms. Shingles preserve local word order, which is exactly the signal a near-duplicate detector needs because reordered or slightly edited text shares most of its shingles.

What You'll Build & Learn

Generate the overlapping k-word shingles of a sentence and count them.

- What a k-shingle is
- Building sliding windows with a range
- Why local word order matters

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 232: detect exact duplicates via sha-1 of text. Project 234 changes the surface problem while preserving the underlying method.

progress check

Reading target: build k-shingles from page text is the surface task; the transferable Python move is to mark setup, rule, and output before running.

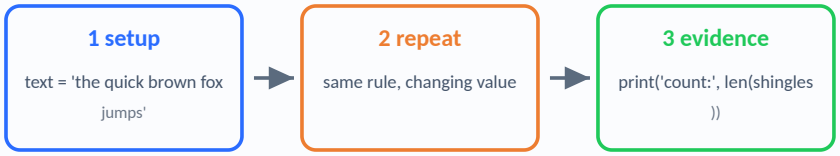
setup -> rule -> result

Logic Lens

- Setup: `text = 'the quick brown fox jumps'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('count:', len(shingles))` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`text = 'the quick brown fox jumps'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`shingles = [tuple(words[i:i + k]) for i in range(len(words) - k + 1)]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('count:', len(shingles))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `shingles = [tuple(words[i:i + k]) for i in range(len(words) - k + 1)]`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build K-Shingles from Page Text: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: text, the, quick, brown, fox. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: text = 'the quick brown fox jumps' | Core: shingles = [tuple(words[i:i + k]) for i in range(len(words) - k + 1)] | Evidence: print('count:', len(shingles))

🛠️ Reader to Writer Bridge

1. Read: explain build k-shingles from page text in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 232.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 234 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('count:', len(shingles))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `text = 'the quick brown fox jumps'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build k-shingles from page text and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A document of n words yields `n - k + 1` overlapping k-word windows; the windows preserve phrasing that a bag-of-words count would discard.

```
text = 'the quick brown fox jumps'
words = text.split()
k = 3
shingles = [tuple(words[i:i + k]) for i in range(len(words) - k + 1)]
for s in shingles:
```

```
    print(' '.join(s))
print('count:', len(shingles))
```

```
output
the quick brown
quick brown fox
brown fox jumps
count: 3
```

How it works:

1. The text is split into a list of individual words
2. k is fixed at three for the window width
3. A comprehension slices words from i to i+k for each valid start index
4. The range stops at len(words) minus k plus one so no window runs off the end
5. Each shingle tuple is printed and the total count reported

Try it:

1. Change k to two and observe how the shingle count grows
2. Build character shingles instead of word shingles
3. Lowercase and strip punctuation before shingling for cleaner matches
4. Change k to 2 and watch the shingle count rise by one.

⚠ Common Mistakes

- Looping to `len(words)` instead of `len(words) - k + 1` produces short, truncated tail shingles.
- Using a list of shingles where order does not matter; a set is better for similarity but allows duplicates here.
- Choosing k larger than the document length yields zero shingles with no error.

Skills gained list slicing | tuple creation | sliding window

→ Where You'll Use This: Shingling is the front end of MinHash, Jaccard and SimHash pipelines; every near-duplicate page detector starts by chopping content into these overlapping word windows.

Mental Model

Reading text through a small window that slides one word at a time, like dragging a magnifying glass across a sentence three words wide.

Project 234 Jaccard Similarity of Two Pages

The Idea Behind It

Jaccard similarity scores how much two pages overlap as the size of their shingle intersection divided by the size of their union, landing between zero and one. The project builds 2-word shingle sets for two near-identical phrases, computes the shared and total shingles with set operators, and rounds the ratio. Because it works on sets, repeated phrases do not inflate the score, making it the classic near-duplicate yardstick.

What You'll Build & Learn

Measure the Jaccard overlap of two short pages from their shingle sets.

- The Jaccard formula intersection over union
- Building shingle sets
- Reading a 0-to-1 similarity score

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 233: build k-shingles from page text. Project 235 changes the surface problem while preserving the underlying method.

progress check

Reading target: jaccard similarity of two pages is the surface task; the transferable Python move is to mark setup, rule, and output before running.

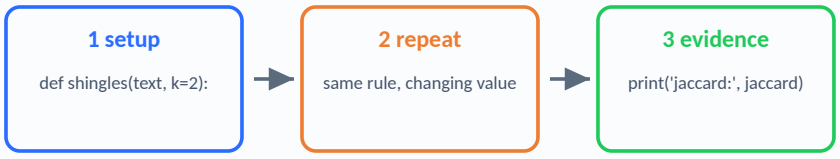
setup -> rule -> result

Logic Lens

- Setup: `def shingles(text, k=2):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('jaccard:', jaccard)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output.

visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def shingles(text, k=2):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`return {tuple(w[i:i + k]) for i in range(len(w) - k + 1)}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('jaccard:', jaccard)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return {tuple(w[i:i + k]) for i in range(len(w) - k + 1)}`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Jaccard Similarity of Two Pages: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: shingles, text, lower, split, tuple. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: def shingles(text, k=2): | Core: return {tuple(w[i:i + k]) for i in range(len(w) - k + 1)} | Evidence: print('jaccard:', jaccard)

🛠️ Reader to Writer Bridge

1. Read: explain jaccard similarity of two pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 233.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 235 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('jaccard:', jaccard)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `def shingles(text, k=2):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of jaccard similarity of two pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Set intersection counts shared shingles and union counts distinct ones; their ratio is Jaccard similarity, which is 1.0 for identical shingle sets and 0.0 for disjoint ones.

```
def shingles(text, k=2):
    w = text.lower().split()
    return {tuple(w[i:i + k]) for i in range(len(w) - k + 1)}

a = shingles('red running shoes for sale')
b = shingles('red running shoes on sale')
inter = a & b
union = a | b
jaccard = round(len(inter) / len(union), 3)
print('shared shingles:', len(inter))
print('total shingles:', len(union))
print('jaccard:', jaccard)
```

```
output
shared shingles: 2
total shingles: 6
jaccard: 0.333
```

How it works:

1. The shingles function lowercases, splits and forms a set of 2-word tuples
2. Two phrases differing in one word are converted to sets a and b
3. The intersection operator finds shared shingles, union finds all distinct ones
4. The Jaccard score divides intersection size by union size, rounded to three places

Try it:

1. Raise k to three and watch the similarity drop for short texts
2. Wrap the calculation in a function taking two strings
3. Guard against an empty union to avoid dividing by zero
4. Make the two strings identical and confirm Jaccard is 1.0.

⚠ Common Mistakes

- Dividing by `len(a) + len(b)` instead of the union double-counts shared shingles.
- Using a list instead of a set makes `&` and `|` unavailable and breaks the maths.
- Forgetting to lowercase lets `Red` and `red` form different shingles.

Skills gained set intersection | set union | jaccard ratio

→ Where You'll Use This: Jaccard is the reference metric behind 'these two pages are 80% the same'; SEO tools quote it when flagging templated or scraped content that barely differs across URLs.

⚡ Mental Model

Two overlapping circles in a Venn diagram: Jaccard is the shared lens divided by the whole figure, a clean fraction of agreement.

Project 235 Near-Duplicate Detection Above a Threshold

💡 The Idea Behind It

Auditing a set for near-duplicates means scoring every pair and reporting the ones that clear a threshold. This project builds 2-word shingle sets per page, then runs a nested loop over the sorted items so each pair is compared once, computing inline Jaccard. Any pair at or above 0.5 prints a NEAR-DUP line. Two pages differing by a single word score high while an unrelated page stays apart.

What You'll Build & Learn

Flag every page pair whose Jaccard similarity meets a near-duplicate threshold.

- Comparing all unique pairs once
- Applying a similarity threshold

- Surfacing near-duplicate clusters

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 234: jaccard similarity of two pages. Project 236 changes the surface problem while preserving the underlying method.

progress check

Reading target: near-duplicate detection above a threshold is the surface task; the transferable Python move is to mark setup, rule, and output before running.

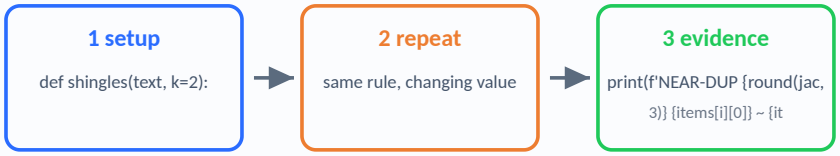
setup -> rule -> result

Logic Lens

- Setup: `def shingles(text, k=2):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'NEAR-DUP {round(jac, 3)} {items[i][0]} ~ {items[j][0]}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output.

visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def shingles(text, k=2):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if jac >= threshold:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'NEAR-DUP {round(jac, 3)} {items[i][0]} ~ {items[j][0]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if jac >= threshold:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Near-Duplicate Detection Above a Threshold: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: shingles, text, lower, split, tuple. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: def shingles(text, k=2): | Core: if jac >= threshold: | Evidence: print(f'NEAR-DUP {round(jac, 3)} {items[i][0]} ~ {items[j][0]}')

Reader to Writer Bridge

1. Read: explain near-duplicate detection above a threshold in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 234.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 236 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'NEAR-DUP {round(jac, 3)} {items[i][0]} ~ {items[j][0]}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `def shingles(text, k=2):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of near-duplicate detection above a threshold and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

An upper-triangular double loop (`j` starts at `i + 1`) compares each unordered pair exactly once, flagging those whose similarity meets the threshold.

```
def shingles(text, k=2):
    w = text.lower().split()
    return {tuple(w[i:i + k]) for i in range(len(w) - k + 1)}

pages = {
    'https://example.com/p1': 'cheap blue widgets for every home',
    'https://example.com/p2': 'cheap blue widgets for every house',
    'https://example.com/p3': 'premium handcrafted oak furniture sets',
}
threshold = 0.5
items = sorted(pages.items())
for i in range(len(items)):
    for j in range(i + 1, len(items)):
        a, b = shingles(items[i][1]), shingles(items[j][1])
        jac = len(a & b) / len(a | b)
        if jac >= threshold:
            print(f'NEAR-DUP {round(jac, 3)} {items[i][0]} ~ {items[j][0]}')
```

```
output
NEAR-DUP 0.667 https://example.com/p1 ~ https://example.com/p2
```

How it works:

1. A shingles helper turns each page body into a set of word-pair tuples
2. Items are sorted so the nested loops produce deterministic, non-repeating pairs
3. The inner loop starts at i+1 to avoid comparing a page with itself or twice
4. Inline Jaccard divides intersection by union for each pair
5. Pairs scoring at or above the 0.5 threshold print a NEAR-DUP report

Try it:

1. Lower the threshold to 0.3 and see more pairs flagged
2. Store flagged pairs in a list instead of printing immediately
3. Skip the O(n squared) sweep by bucketing pages first for speed
4. Lower the threshold to 0.3 and watch p3 start matching.

⚠ Common Mistakes

- Starting the inner loop at `j = 0` compares a page to itself (Jaccard 1.0) and reports every pair twice.
- Setting the threshold too low floods the report with unrelated pairs.
- Sorting items is needed for deterministic output; iterating the raw dict relies on insertion order.

Skills gained nested loops | pairwise comparison | threshold filter

→ Where You'll Use This: Site-wide duplicate reports run this pairwise sweep to surface clusters of barely-edited pages, the kind that dilute crawl budget and split ranking signals across copies.

Mental Model

Like a teacher comparing every essay against every other for copied passages, raising a flag whenever two read too much alike.

Project 236 Flag Thin Content by Word Count

The Idea Behind It

A page with too few words rarely answers a searcher's question and reads as low value to an engine. This project counts the word-like tokens in each page body using a regex that matches runs of letters and apostrophes, then tags any page below a ten-word threshold as THIN. The regex deliberately ignores punctuation-only fragments so 'Buy now.' counts as two real words, not noise.

What You'll Build & Learn

Flag pages whose word count falls below a thin-content threshold.

- Counting word tokens with regex
- Applying a thin-content threshold
- Why short pages underperform

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 235: near-duplicate detection above a threshold. Project 237 changes the surface problem while preserving the underlying method. *progress check*

Reading target: flag thin content by word count is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'{tag} {wc} words {url}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`tag = 'THIN' if wc < threshold else 'ok'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{tag} {wc} words {url}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `tag = 'THIN' if wc < threshold else 'ok'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Thin Content by Word Count: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: re, pages, https, example, com. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: tag = 'THIN' if wc < threshold else 'ok' | Evidence: print(f'{tag} {wc} words {url}')

🛠️ Reader to Writer Bridge

1. Read: explain flag thin content by word count in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 235.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 237 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{tag} {wc} words {url}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag thin content by word count and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A regex for word-like tokens counts real words while ignoring punctuation, and comparing that count to a threshold flags pages too thin to rank.

```
import re
pages = {
    'https://example.com/full': 'Our complete guide explains every step of
installing solar panels safely at home this season.',
    'https://example.com/thin': 'Solar panels. Buy now.',
    'https://example.com/ok': 'Solar panels convert sunlight into electricity
using photovoltaic cells mounted on your roof.',
}
threshold = 10
for url in sorted(pages):
    wc = len(re.findall(r"[a-zA-Z']+", pages[url]))
    tag = 'THIN' if wc < threshold else 'ok'
    print(f'{tag} {wc} words {url}')
```

```
output
ok 15 words https://example.com/full
ok 13 words https://example.com/ok
THIN 4 words https://example.com/thin
```

How it works:

1. A dictionary maps three URLs to bodies of varying length
2. re.findall with a letters-and-apostrophe pattern extracts word tokens
3. len of that list gives the word count per page
4. Pages under the ten-word threshold are tagged THIN, the rest ok
5. Output prints the tag, count and URL for each sorted page

Try it:

1. Raise the threshold to fifty words for long-form requirements
2. Count sentences as well and require a minimum of both
3. Exclude boilerplate words before counting to measure unique content
4. Raise the threshold to 14 and watch the `ok` page flip to THIN.

⚠ Common Mistakes

- Splitting on whitespace counts `'panels.'` and `'now.'` with attached punctuation as words.
- Counting characters instead of words penalises pages with long words unfairly.
- A fixed threshold ignores page type: a contact page is legitimately short.

Skills gained re.findall | word counting | threshold tagging

→ Where You'll Use This: Thin-content reports rank pages by word count to find doorway, placeholder or auto-generated URLs that should be expanded, merged, or pruned from the index.

Mental Model

A word-count gate is the bouncer at a venue checking nobody slips in with an empty page where real content should be.

Project 237 Compute a Boilerplate-to-Content Ratio

The Idea Behind It

Templated navigation and footer terms like home, login and privacy repeat on every page, and when they crowd out the real content the page is effectively thin. This project keeps a set of boilerplate words, counts how many of a page's tokens fall in that set, and divides by the total to get a ratio. When boilerplate exceeds half the words, the unique content is too sparse to rank.

What You'll Build & Learn

Compute the share of a page's words that are boilerplate and judge it accordingly.

- Defining a boilerplate word set
- Computing a boilerplate ratio
- Distinguishing template from content

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 236: flag thin content by word count. Project 238 changes the surface problem while preserving the underlying method.

progress check

Reading target: compute a boilerplate-to-content ratio is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

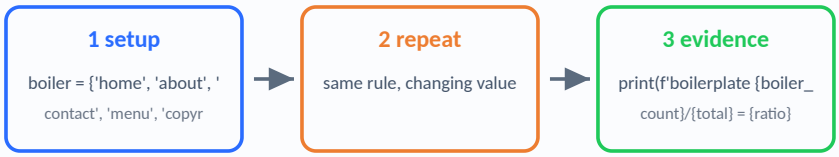
Logic Lens

- Setup: `boiler = {'home', 'about', 'contact', 'menu', 'copyright', 'privacy', 'login'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the

later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'boilerplate {boiler_count}/{total} = {ratio} -> {verdict}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`boiler = {'home', 'about', 'contact', 'menu', 'copyright', 'privacy', 'login'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`boiler_count = sum(1 for w in page_words if w in boiler)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'boilerplate {boiler_count}/{total} = {ratio} -> {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `boiler_count = sum(1 for w in page_words if w in boiler)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute a Boilerplate-to-Content Ratio: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: boiler, home, about, contact, menu. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: boiler = {'home', 'about', 'contact', 'menu', 'copyright', 'privacy', '... | Core: boiler_count = sum(1 for w in page_words if w in boiler) | Evidence: print(f'boilerplate {boiler_count}/{total} = {ratio} -> {verdict}')

🛠️ Reader to Writer Bridge

1. Read: explain compute a boilerplate-to-content ratio in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 236.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 238 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'boilerplate {boiler_count}/{total} = {ratio} -> {verdict}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `boiler = {'home', 'about', 'contact', 'menu', 'copyright', 'privacy', 'login'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute a boilerplate-to-content ratio and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Counting how many page words appear in a known boilerplate set, divided by the total, estimates how much of a page is template versus unique content.

```
boiler = {'home', 'about', 'contact', 'menu', 'copyright', 'privacy', 'login'}
page_words = ['home', 'menu', 'login', 'solar', 'panels', 'convert',
'sunlight', 'electricity', 'copyright', 'privacy']
total = len(page_words)
boiler_count = sum(1 for w in page_words if w in boiler)
ratio = round(boiler_count / total, 3)
verdict = 'THIN (boilerplate-heavy)' if ratio > 0.5 else 'ok'
print(f'boilerplate {boiler_count}/{total} = {ratio} -> {verdict}')
```

```
output
boilerplate 5/10 = 0.5 -> ok
```

How it works:

1. A set holds common nav and footer terms for fast membership tests
2. A list represents the page's tokenised words mixing boilerplate and content
3. A generator counts how many words are in the boilerplate set
4. The ratio divides that count by the total word count, rounded
5. A threshold above 0.5 yields a boilerplate-heavy verdict

Try it:

1. Expand the boilerplate set with real header and footer phrases
2. Report the count of unique non-boilerplate words instead of a ratio
3. Lower the threshold and rescore the same page
4. Add `'footer'` and `'search'` to the boilerplate set and recompute.

⚠ Common Mistakes

- Using `>=` instead of `>` would flip this exact 0.5 case to THIN; pick the boundary deliberately.
- A tiny boilerplate set undercounts real template words and hides thin pages.

- Counting unique words instead of all occurrences ignores boilerplate that repeats.

Skills gained set membership generator sum ratio threshold

→ Where You'll Use This: Measuring boilerplate-to-content ratio helps separate genuinely useful pages from skeletal templates whose body is mostly menus, a frequent cause of thin-content flags at scale.

Mental Model

Like measuring how much of a newspaper page is ads versus story: when the ads take over, there's barely an article left to read.

Project 238 Find Duplicate Titles Across a Set

The Idea Behind It

Two pages sharing the same title tag compete in the search results and muddy which one the engine should rank. This project groups URLs by their exact title string using setdefault, then prints any title that maps to more than one page. The three pages titled 'Best Running Shoes' collapse into one group, instantly revealing the clash that a single-page check would never catch.

What You'll Build & Learn

Group pages by title and report any title shared across more than one URL.

- Grouping records by a shared key
- Detecting duplicate title tags
- Why duplicate titles hurt ranking

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 237: compute a boilerplate-to-content ratio. Project 239 changes the surface problem while preserving the underlying method.

progress check

Reading target: find duplicate titles across a set is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

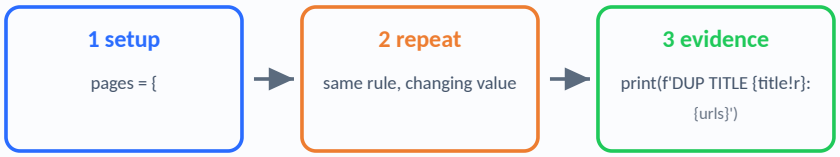
Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'DUP TITLE {title!r}: {urls}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(urls) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'DUP TITLE {title!r}: {urls}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(urls) > 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Duplicate Titles Across a Set: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, https, example, com, Best. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = { | Core: if len(urls) > 1: | Evidence: print(f'DUP TITLE {title!r}: {urls}')

🛠️ Reader to Writer Bridge

1. Read: explain find duplicate titles across a set in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 237.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 239 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'DUP TITLE {title!r}: {urls}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of find duplicate titles across a set and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A dict keyed by title collects every URL sharing it; groups of size greater than one are the duplicate-title clashes to fix.

```
pages = {
    'https://example.com/a': 'Best Running Shoes',
    'https://example.com/b': 'Best Running Shoes',
    'https://example.com/c': 'Trail Running Guide',
    'https://example.com/d': 'Best Running Shoes',
}
groups = {}
for url, title in pages.items():
    groups.setdefault(title, []).append(url)
for title in sorted(groups):
    urls = sorted(groups[title])
    if len(urls) > 1:
        print(f'DUP TITLE {title!r}: {urls}')
```

```
output
DUP TITLE 'Best Running Shoes': ['https://example.com/a', 'https://example.com/b',
'https://example.com/d']
```

How it works:

1. A dictionary maps four URLs to their title strings
2. setdefault appends each URL to a list keyed by its title
3. Sorting the group keys gives deterministic output order
4. Any title with more than one URL prints a DUP TITLE line with the sorted list

Try it:

1. Normalise titles by case and whitespace before grouping
2. Also report near-identical titles differing by one word
3. Count the total number of clashing pages across all titles
4. Lowercase and strip each title before grouping to catch near-identical clashes.

⚠ Common Mistakes

- Comparing titles case-sensitively misses `'best running shoes'` versus `'Best Running Shoes'`.
- Not trimming whitespace lets a trailing space split an otherwise identical title.
- Reporting every group instead of only those with more than one URL spams the unique titles.

Skills gained `setdefault grouping` `duplicate detection` `title audit`

→ **Where You'll Use This:** Duplicate-title reports are a staple of on-page audits; identical titles across product or category pages signal templating bugs and self-competition that depress click-through.

⚡ Mental Model

Like two shops on one street with the exact same sign: shoppers and search engines can't tell which door to walk through.

Project 239 Collapse www vs Non-www Duplicates

💡 The Idea Behind It

A site answering at both www and the bare host serves every page twice under two addresses. This project treats the www prefix as cosmetic, stripping it with a single targeted replace so both variants map to the same key, then groups the original URLs under that key. Any key holding more than one variant is a www duplicate, and the fix is to pick one host and redirect the other.

What You'll Build & Learn

Detect URLs that differ only by the www prefix and group them as duplicates.

- Treating www as a cosmetic prefix
- Folding host variants to one key
- Why both hosts cause duplication

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 238: find duplicate titles across a set. Project 240 changes the surface problem while preserving the underlying method.

progress check

Reading target: collapse www vs non-www duplicates is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `urls = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.

- Evidence: `print(f'WWW DUP {canon}: {variants}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(variants) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'WWW DUP {canon}: {variants}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(variants) > 1:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Collapse www vs Non-www Duplicates: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: urls, https, www, example, com. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: urls = [| Core: if len(variants) > 1: | Evidence: print(f'WWW DUP {canon}: {variants}')

🛠️ Reader to Writer Bridge

1. Read: explain collapse www vs non-www duplicates in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 238.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 240 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'WWW DUP {canon}: {variants}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `urls = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of collapse www vs non-www duplicates and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Replacing the first `://www.` with `://` maps both host variants to one key, so grouping reveals which paths exist under both.

```
urls = [
    'https://www.example.com/blog',
    'https://example.com/blog',
    'https://www.example.com/about',
    'https://example.com/contact',
]
def strip_www(u):
    return u.replace('://www.', '://', 1)
groups = {}
for u in urls:
    groups.setdefault(strip_www(u), []).append(u)
for canon in sorted(groups):
    variants = sorted(groups[canon])
    if len(variants) > 1:
        print(f'WWW DUP {canon}: {variants}')
```

```
output
WWW DUP https://example.com/blog: ['https://example.com/blog',
'https://www.example.com/blog']
```

How it works:

1. A list holds URLs mixing www and bare-host spellings
2. strip_www replaces the www. prefix once to produce the folded key
3. setdefault collects original URLs under each stripped key
4. Keys with more than one variant print a WWW DUP report with both spellings

Try it:

1. Handle URLs that already lack a scheme gracefully
2. Choose a canonical host and emit the redirect that should exist
3. Combine this with trailing-slash folding in one pass
4. Decide `www.` is canonical and rewrite bare hosts to add it instead.

⚠ Common Mistakes

- Using `.replace('www.', '')` without the `://` anchor would also strip `www.` from a path or query.
- Passing no count argument to `replace` could rewrite a later `://www.` in a redirect target.
- Assuming the bare host is always canonical; some sites canonicalise to `www.` instead.

Skills gained str.replace | url folding | host normalization

→ Where You'll Use This: The www-versus-non-www split is one of the oldest duplicate-content traps; auditors confirm one host redirects to the other so ranking signals don't divide across two front doors.

Mental Model

www and the bare domain are two doorbells on the same house; press either and the same family answers, but visitors think there are two homes.

Project 240 Detect a Page Served Over Two Protocols

The Idea Behind It

When one page answers on both HTTPS and HTTP it is a single page living under two URLs. This project ignores the scheme as content and groups by the host-and-path, collecting the set of protocols seen for each. A host-path that appears under more than one protocol is flagged as a duplicate, with the recommended fix being to force HTTPS and redirect the insecure variant away.

What You'll Build & Learn

Find a host-and-path served over both protocols and recommend forcing HTTPS.

- Grouping by host-and-path across schemes
- Collecting protocols in a set
- Why mixed protocols duplicate a page

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 239: collapse www vs non-www duplicates. Project 241 changes the surface problem while preserving the underlying method.

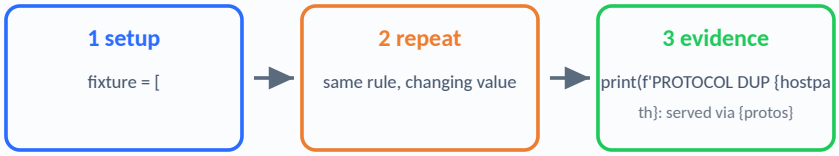
progress check

Reading target: detect a page served over two protocols is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `fixture = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'PROTOCOL DUP {hostpath}: served via {protos} (canonicalise to HTTPS)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`fixture = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(protos) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'PROTOCOL DUP {hostpath}: served via {protos} (canonicalise to HTTPS)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(protos) > 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Page Served Over Two Protocols: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: fixture, example, com, login, home. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: fixture = [| Core: if len(protos) > 1: | Evidence: print(f'PROTOCOL DUP {hostpath}: served via {protos} (canonicalise to H...

🛠️ Reader to Writer Bridge

1. Read: explain detect a page served over two protocols in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 239.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 241 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'PROTOCOL DUP {hostpath}: served via {protos} (canonicalise to HTTPS)')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `fixture = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a page served over two protocols and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Keying a set of protocols by host-and-path reveals any path answering on more than one protocol; the fix is to redirect every request to the HTTPS variant.

```
fixture = [
    ('HTTPS', 'example.com/login'),
    ('HTTP', 'example.com/login'),
    ('HTTPS', 'example.com/home'),
```

```
]
groups = {}
for proto, hostpath in fixture:
    groups.setdefault(hostpath, set()).add(proto)
for hostpath in sorted(groups):
    protos = sorted(groups[hostpath])
    if len(protos) > 1:
        print(f'PROTOCOL DUP {hostpath}: served via {protos} (canonicalise to
HTTPS)')
```

```
output
PROTOCOL DUP example.com/login: served via ['HTTP', 'HTTPS'] (canonicalise to HTTPS)
```

How it works:

1. A fixture lists protocol and hostpath tuples for three observations
2. setdefault keyed by hostpath gathers the protocols into a set
3. Sorting the keys gives stable output
4. Any hostpath whose protocol set has more than one entry prints a PROTOCOL DUP line

Try it:

1. Add a port to the host-path and fold default ports too
2. Report which protocol is missing for single-protocol pages
3. Emit the exact redirect rule needed to force HTTPS
4. Add `('HTTP', 'example.com/home')` so the home page also flags.

⚠ Common Mistakes

- Including the protocol in the grouping key hides the duplicate by making each URL look distinct.
- Using a list instead of a set could record the same protocol twice and miscount.
- Forgetting to sort the protocols makes the printed pair order nondeterministic.

Skills gained set accumulation | tuple unpacking | protocol audit

→ Where You'll Use This: Mixed HTTP and HTTPS serving is a classic migration leftover; audits confirm the secure version is canonical and the insecure one redirects, consolidating both into one indexed URL.

Mental Model

Like a house with two address plates, one old and one new; the postman should only ever deliver to the new, secure one.

Project 241 Classify Pagination with rel next and prev

The Idea Behind It

A paginated listing links each page to its neighbours through rel-next and rel-prev pointers, and reading them reveals each page's role in the sequence. This project models three list pages as dicts with prev and next fields, then classifies each: a missing prev means first, a missing next means last, and having both means middle. The series structure falls straight out of which pointers are present.

What You'll Build & Learn

Classify each page of a paginated series as first, middle or last from its pointers.

- Reading rel-next and rel-prev pointers
- Inferring position from None pointers
- Modelling a pagination series

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 240: detect a page served over two protocols. Project 242 changes the surface problem while preserving the underlying method.

progress check

Reading target: classify pagination with rel next and prev is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

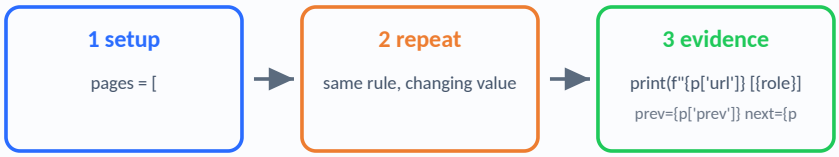
- Setup: `pages = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f"{p['url']} [{role}] prev={p['prev']} next={p['next']}")` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`role = 'first' if p['prev'] is None else ('last' if p['next'] is None else 'middle')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"{p['url']} [{role}] prev={p['prev']} next={p['next']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `role = 'first' if p['prev'] is None else ('last' if p['next'] is None else 'middle')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Classify Pagination with rel next and prev: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, url, https, example, com. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = [| Core: role = 'first' if p['prev'] is None else ('last' if p['next'] is None e... | Evidence: print(f"{p['url']} [{role}] prev={p['prev']} next={p['next']}")

🛠️ Reader to Writer Bridge

1. Read: explain classify pagination with rel next and prev in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 240.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 242 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{p['url']} [{role}] prev={p['prev']} next={p['next']}")`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `pages = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify pagination with rel next and prev and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A missing prev marks the first page and a missing next marks the last; everything with both is a middle page, which is how a series is anchored without canonicalising every page to page one.

```
pages = [
    {'url': 'https://example.com/list?page=1', 'prev': None, 'next':
'https://example.com/list?page=2'},
    {'url': 'https://example.com/list?page=2', 'prev':
'https://example.com/list?page=1', 'next': 'https://example.com/list?page=3'},
```

```
    {'url': 'https://example.com/list?page=3', 'prev':
'https://example.com/list?page=2', 'next': None},
]
for p in pages:
    role = 'first' if p['prev'] is None else ('last' if p['next'] is None else
'middle')
    print(f"{p['url']} [{role}] prev={p['prev']} next={p['next']}")
```

```
output
https://example.com/list?page=1 [first] prev=None next=https://example.com/list?
page=2
https://example.com/list?page=2 [middle] prev=https://example.com/list?page=1
next=https://example.com/list?page=3
https://example.com/list?page=3 [last] prev=https://example.com/list?page=2
next=None
```

How it works:

1. A list of dicts gives each list page a url, prev and next field
2. The loop assigns role first when prev is None, last when next is None, else middle
3. A nested ternary expresses the three-way classification
4. An f-string prints the URL, its role and both neighbour pointers

Try it:

1. Validate that each next pointer matches the following page's prev
2. Detect a broken series where a middle page has no prev
3. Build the full ordered chain from any starting page
4. Add a page 4 and confirm page 3 becomes a middle page.

⚠ Common Mistakes

- Pointing every page's canonical at page 1 instead of using rel-next/prev orphans the deeper pages.
- Treating an empty string as different from `None` breaks the first/last test.
- Assuming the list is ordered; a real crawl may see page 3 before page 1.

Skills gained None checks | nested ternary | pagination logic

→ Where You'll Use This: Understanding pagination roles matters when deciding canonical and indexing strategy for category and archive pages, where mishandling deep pages can hide content from search entirely.

⚡ Mental Model

Pagination pointers are train cars coupled front and back; the engine that has no car behind it is the locomotive, the one with none ahead is the caboose.

Project 242 Count the Faceted-Navigation URL Explosion

The Idea Behind It

Filterable shop facets multiply combinatorially: each facet can be set to one of its values or left unset, so the number of crawlable URLs is the product of value-counts-plus-one across all facets. This project sorts the facet keys, multiplies those plus-one factors together, and prints the total alongside one example URL. Three small facets already explode into thirty-six near-identical listing pages.

What You'll Build & Learn

Count the URL variants a set of navigation facets produces and show an example.

- Why facets multiply combinatorially
- The plus-one for the unset state
- Crawl-budget impact of facets

Skill Roadmap

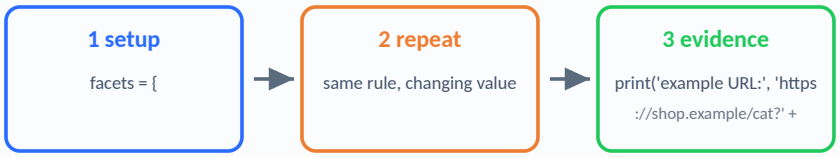
Skill step: moves you from recognition toward independent tracing. It builds on Project 241: classify pagination with rel next and prev. Project 243 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count the faceted-navigation url explosion is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `facets = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('example URL:', 'https://shop.example/cat?' + example)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`facets = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`for k in keys:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('example URL:', 'https://shop.example/cat?' + example)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for k in keys:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count the Faceted-Navigation URL Explosion: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: facets, color, red, blue, size. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: facets = { | Core: for k in keys: | Evidence: print('example URL:', 'https://shop.example/cat?' + example)

🛠️ Reader to Writer Bridge

1. Read: explain count the faceted-navigation url explosion in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 241.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 243 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('example URL:', 'https://shop.example/cat?' + example)`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `facets = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count the faceted-navigation url explosion and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each facet contributes its number of values plus one (the unset state), and multiplying across facets gives the total crawlable combinations, the count to control with noindex or robots rules.

```
facets = {
    'color': ['red', 'blue'],
    'size': ['s', 'm', 'l'],
    'brand': ['acme', 'globex'],
}
```

```
keys = sorted(facets)
combos = 1
for k in keys:
    combos *= (len(facets[k]) + 1)
print('facets:', keys)
print('URL variants (incl. none-selected):', combos)
example = '&'.join(f'{k}={facets[k][0]}' for k in keys)
print('example URL:', 'https://shop.example/cat?' + example)
```

```
output
facets: ['brand', 'color', 'size']
URL variants (incl. none-selected): 36
example URL: https://shop.example/cat?brand=acme&color=red&size=s
```

How it works:

1. A dict maps each facet to its list of possible values
2. Keys are sorted for deterministic output
3. combos starts at one and is multiplied by each facet's value count plus one for unset
4. The total is printed, then an example URL is built from each facet's first value

Try it:

1. Add a fourth facet and watch the total balloon
2. Allow multi-select facets where values combine, raising the count further
3. Print every generated URL for a tiny facet set to see them all
4. Add a fourth facet and watch the count multiply again.

⚠ Common Mistakes

- Multiplying by `len(values)` without the `+1` forgets the unset state of each facet.
- Summing facet sizes instead of multiplying drastically understates the explosion.
- Ignoring parameter order: without canonicalisation each ordering is yet another URL.

Skills gained product accumulation | combinatorics | facet modelling

→ Where You'll Use This: Faceted navigation is a top cause of index bloat on e-commerce sites; estimating the explosion justifies robots rules, canonical tags or parameter handling before crawlers drown in filter combinations.

Mental Model

Each facet is a light switch; with switches that also have an off position, the number of room states multiplies fast into a wall of crawlable pages.

Project 243 Fold Parameter-Order Duplicate URLs

The Idea Behind It

Query parameters listed in a different order point at the very same page, yet a naive crawler treats every ordering as a distinct URL. This project canonicalises each URL by splitting off the query, sorting the parameters alphabetically, and rejoining them, then groups originals under that canonical key. Three reorderings of a=1, b=2, c=3 collapse into one group, exposing them as a single page.

What You'll Build & Learn

Group URLs that differ only in parameter order under one sorted-query canonical.

- Why parameter order does not change a page
- Sorting query params for canonicalisation
- Grouping by canonical key

Skill Roadmap

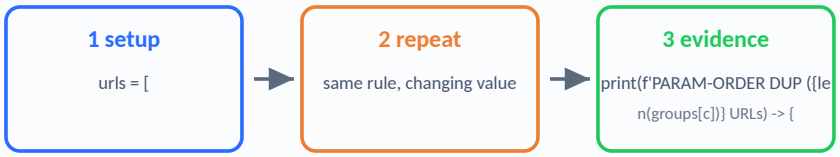
Skill step: moves you from recognition toward independent tracing. It builds on Project 242: count the faceted-navigation url explosion. Project 244 changes the surface problem while preserving the underlying method. *progress check*

Reading target: fold parameter-order duplicate urls is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `urls = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'PARAM-ORDER DUP ({len(groups[c])} URLs) -> {c}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`params = '&'.join(sorted(query.split('&')) ) if query else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'PARAM-ORDER DUP ({len(groups[c])} URLs) -> {c}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `params = '&'.join(sorted(query.split('&'))) if query else ''`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Fold Parameter-Order Duplicate URLs: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: urls, https, shop, example, canon. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: urls = [| Core: params = '&'.join(sorted(query.split('&'))) if query else '' | Evidence: print(f'PARAM-ORDER DUP ({len(groups[c])} URLs) -> {c}')

🛠️ Reader to Writer Bridge

1. Read: explain fold parameter-order duplicate urls in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 242.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 244 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'PARAM-ORDER DUP ({len(groups[c])} URLs) -> {c}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `urls = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of fold parameter-order duplicate urls and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Sorting the `&`-separated params produces an order-independent canonical query, so URLs differing only in parameter order collapse to one group.

```
urls = [
    'https://shop.example/p?a=1&b=2&c=3',
    'https://shop.example/p?c=3&a=1&b=2',
    'https://shop.example/p?b=2&a=1&c=3',
    'https://shop.example/q?a=1',
]
```

```
def canon(u):
    base, _, query = u.partition('?')
    params = '&'.join(sorted(query.split('&'))) if query else ''
    return base + ('?' + params if params else '')
groups = {}
for u in urls:
    groups.setdefault(canon(u), []).append(u)
for c in sorted(groups):
    if len(groups[c]) > 1:
        print(f'PARAM-ORDER DUP ({len(groups[c])} URLs) -> {c}')
```

```
output
PARAM-ORDER DUP (3 URLs) -> https://shop.example/p?a=1&b=2&c=3
```

How it works:

1. A list holds three reorderings of one query plus an unrelated URL
2. canon partitions off the query and sorts the ampersand-separated params
3. It rebuilds the URL with the sorted query, or none if empty
4. setdefault groups originals under each canonical form
5. Groups with more than one URL print a PARAM-ORDER DUP line and the count

Try it:

1. Also strip tracking parameters before sorting
2. Lowercase parameter keys before comparison
3. Report the canonical winner you would keep for each group
4. Add a fourth ordering of the same params and watch the group grow to 4.

⚠ Common Mistakes

- Sorting the whole query string as characters rather than splitting on `&` scrambles the params.
- Not handling the no-query case leaves a trailing `?` on parameterless URLs.
- Assuming param order never matters; for some apps it does and folding breaks them.

Skills gained str.partition | param sorting | url grouping

→ Where You'll Use This: Parameter-order duplication is invisible to humans but real to crawlers; canonicalising by sorted query lets a dedupe pipeline recognise these as one page and consolidate them.

Mental Model

Like three people listing the same grocery items in a different order; sort each list and you see they were buying identical baskets.

Project 244 Fold Trailing-Slash Duplicate URLs

The Idea Behind It

A URL with a trailing slash and one without usually serve the same page, so they should fold into a single canonical. This project parses the URL into host and path, ensures a leading slash, and strips a trailing slash from any path longer than the root, then groups originals under that canonical key. The blog and blog-with-slash pair collapses while the root slash is correctly left alone.

What You'll Build & Learn

Collapse URLs differing only by a trailing slash on non-root paths.

- Trailing slashes as cosmetic on non-root paths
- Preserving the root slash
- Folding to one canonical

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 243: fold parameter-order duplicate urls. Project 245 changes the surface problem while preserving the underlying method.

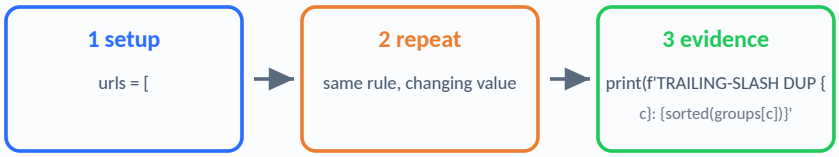
progress check

Reading target: fold trailing-slash duplicate urls is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `urls = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'TRAILING-SLASH DUP {c}: {sorted(groups[c])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(path) > 1 and path.endswith('/'):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'TRAILING-SLASH DUP {c}: {sorted(groups[c])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(path) > 1 and path.endswith('/'):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Fold Trailing-Slash Duplicate URLs: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: urls, https, example, com, blog. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: urls = [| Core: if len(path) > 1 and path.endswith('/'): | Evidence: print(f'TRAILING-SLASH DUP {c}: {sorted(groups[c])}')

🛠️ Reader to Writer Bridge

1. Read: explain fold trailing-slash duplicate urls in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 243.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 245 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'TRAILING-SLASH DUP {c}: {sorted(groups[c])}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `urls = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of fold trailing-slash duplicate urls and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The `len(path) > 1` guard strips a trailing slash from real paths while leaving the root `'/'` intact, so `/blog` and `/blog/` fold but the homepage does not vanish.

```
urls = [
    'https://example.com/blog',
    'https://example.com/blog/',
    'https://example.com/about/',
    'https://example.com/',
]
```

```
def canon(u):
    scheme, rest = u.split('://', 1)
    host, slash, path = rest.partition('/')
    path = '/' + path
    if len(path) > 1 and path.endswith('/'):
        path = path[:-1]
    return scheme + '://' + host + path
groups = {}
for u in urls:
    groups.setdefault(canon(u), []).append(u)
for c in sorted(groups):
    if len(groups[c]) > 1:
        print(f'TRAILING-SLASH DUP {c}: {sorted(groups[c])}')
```

```
output
TRAILING-SLASH DUP https://example.com/blog: ['https://example.com/blog',
'https://example.com/blog/']
```

How it works:

1. A list mixes slashed and unslashed paths including the bare root
2. canon splits scheme from host, then partitions host from path
3. The path is rebuilt with a leading slash for consistency
4. A trailing slash is trimmed only when the path is longer than one character
5. setdefault groups originals, and multi-member groups print a TRAILING-SLASH DUP line

Try it:

1. Make the policy add a slash rather than remove one and rerun
2. Combine with case-folding of the host in the same function
3. Handle a path that is a single slash plus a query string
4. Add `https://example.com/about` (no slash) and watch the about pair fold.

⚠ Common Mistakes

- Stripping the slash unconditionally turns the root `'/'` into an empty path.
- Lowercasing the path here would wrongly merge case-distinct pages.
- Assuming slash and no-slash always match; some servers serve different content.

Skills gained str.partition | path normalization | slash handling

→ Where You'll Use This: Trailing-slash inconsistency splits link equity and crawl signals between two URLs; audits enforce one convention site-wide and redirect the other, which this folding logic models exactly.

Mental Model

A trailing slash is like a period at the end of a directory name; with or without it, you've arrived at the same address.

Project 245 Fingerprint a Page with MinHash-Lite

The Idea Behind It

MinHash estimates Jaccard similarity cheaply by hashing every shingle under several salts and keeping the minimum value per salt. This project builds 2-word shingles, then for each of four salts computes the smallest salted SHA-1 hash modulo a thousand to form a compact signature. Comparing two signatures slot by slot estimates the overlap without ever intersecting the full shingle sets, the trick that makes web-scale dedupe feasible.

What You'll Build & Learn

Build a small MinHash signature for a page and estimate similarity from slot agreement.

- MinHash estimates Jaccard cheaply
- One minimum hash per salt
- Comparing signatures slot by slot

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 244: fold trailing-slash duplicate urls. Project 246 changes the surface problem while preserving the underlying method.

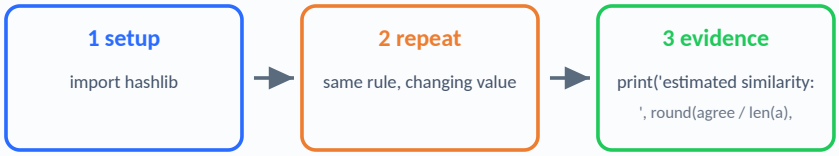
progress check

Reading target: fingerprint a page with minhash-lite is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import hashlib` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('estimated similarity:', round(agree / len(a), 3))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import hashlib`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`agree = sum(1 for x, y in zip(a, b) if x == y)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('estimated similarity:', round(agree / len(a), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `agree = sum(1 for x, y in zip(a, b) if x == y)`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Fingerprint a Page with MinHash-Lite: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: hashlib, minhash, text, salts, s1. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import hashlib | Core: agree = sum(1 for x, y in zip(a, b) if x == y) | Evidence: print('estimated similarity:', round(agree / len(a), 3))

🛠️ Reader to Writer Bridge

1. Read: explain fingerprint a page with minhash-lite in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 244.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 246 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('estimated similarity:', round(agree / len(a), 3))`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import hashlib` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of fingerprint a page with minhash-lite and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each salt acts as one hash permutation; the minimum hashed shingle under that salt is a sampled feature, and the fraction of matching signature slots estimates Jaccard similarity.

```
import hashlib

def minhash(text, salts=('s1', 's2', 's3', 's4')):
    shingles = set()
```

```
    w = text.lower().split()
    for i in range(len(w) - 1):
        shingles.add(w[i] + ' ' + w[i + 1])
    sig = []
    for salt in salts:
        mn = min(int(hashlib.sha1((salt + s).encode()).hexdigest(), 16) for s
in shingles)
        sig.append(mn % 1000)
    return sig

a = minhash('the fast red fox runs')
b = minhash('the fast red fox jumps')
agree = sum(1 for x, y in zip(a, b) if x == y)
print('sig A:', a)
print('sig B:', b)
print('estimated similarity:', round(agree / len(a), 3))
```

```
output
sig A: [999, 101, 152, 867]
sig B: [999, 589, 152, 867]
estimated similarity: 0.75
```

How it works:

1. The minhash function builds a set of space-joined word-pair shingles
2. For each salt it hashes salt-plus-shingle and keeps the minimum integer
3. Each minimum is taken modulo a thousand to keep the signature compact
4. Two near-identical phrases produce signatures of four slots each
5. Counting matching slots and dividing by the length estimates similarity

Try it:

1. Increase the number of salts and watch the estimate steady
2. Compare an unrelated phrase and confirm fewer slots match
3. Use longer signatures and measure accuracy versus speed
4. Add four more salts and watch the estimate steady.

⚠ Common Mistakes

- Reusing one salt for every slot makes all signature positions identical and uninformative.
- Hashing without `.encode()` raises a TypeError on the `str` argument.
- Too few salts make the similarity estimate coarse and jumpy.

Skills gained `hashlib.sha1` `min over hashes` `signature compare`

→ **Where You'll Use This:** MinHash signatures let search engines compare billions of documents without pairwise set math; the same trick powers practical near-duplicate clustering in large-scale SEO tooling.

Mental Model

Like polling a few random judges instead of the whole crowd: ask enough of them and the small sample's agreement closely predicts the full vote.

Project 246 Cluster Duplicate URLs to One Canonical

The Idea Behind It

A complete canonicaliser stacks every folding rule into one function: lowercase the host, drop a leading www, cut the fragment, strip utm tracking parameters, sort what remains, lowercase the path, and trim a trailing slash. This project applies all of that to a messy URL set and groups originals under the resulting key, so three differently-spelled variants of one page collapse into a single cluster while the genuinely distinct URL stands alone.

What You'll Build & Learn

Apply every normalisation rule at once to cluster duplicate URLs to one canonical.

- Composing many normalisation rules
- Order of canonicalisation steps
- Clustering originals under one key

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 245: fingerprint a page with minhash-lite. Project 247 changes the surface problem while preserving the underlying method.

progress check

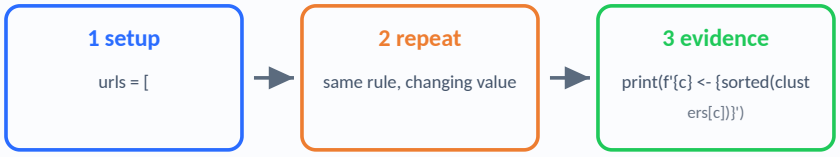
Reading target: cluster duplicate urls to one canonical is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `urls = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.

- Evidence: `print(f'{c} <- {sorted(clusters[c])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`urls = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if host.startswith('www.'):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{c} <- {sorted(clusters[c])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if host.startswith('www.'):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Cluster Duplicate URLs to One Canonical: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: urls, https, www, Example, com. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: urls = [| Core: if host.startswith('www.'): | Evidence: print(f'{c} <- {sorted(clusters[c])}')

🛠️ Reader to Writer Bridge

1. Read: explain cluster duplicate urls to one canonical in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 245.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 247 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c} <- {sorted(clusters[c])}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `urls = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of cluster duplicate urls to one canonical and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Applying host-lowercasing, www-stripping, fragment and tracking-param removal, slash-trimming, and path-lowercasing in one function makes every cosmetic variant produce the same key, so the dict groups them

automatically.

```
urls = [
    'https://www.Example.com/Page/',
    'https://example.com/page',
    'https://example.com/page?utm_source=x',
    'https://example.com/other',
]
def canon(u):
    scheme, rest = u.split('://', 1)
    rest = rest.split('#', 1)[0]
    hostpath, _, query = rest.partition('?')
    host, slash, path = hostpath.partition('/')
    host = host.lower()
    if host.startswith('www.'):
        host = host[4:]
    path = '/' + path
    if len(path) > 1 and path.endswith('/'):
        path = path[:-1]
    kept = [kv for kv in query.split('&') if kv and not kv.startswith('utm_')]
    q = '?' + '&'.join(sorted(kept)) if kept else ''
    return scheme + '://' + host + path.lower() + q
clusters = {}
for u in urls:
    clusters.setdefault(canon(u), []).append(u)
for c in sorted(clusters):
    print(f'{c} <- {sorted(clusters[c])}')
```

```
output
https://example.com/other <- ['https://example.com/other']
https://example.com/page <- ['https://example.com/page', 'https://example.com/page?
utm_source=x', 'https://www.Example.com/Page/']
```

How it works:

1. canon splits scheme, drops the fragment after a hash, and partitions host from query
2. It lowercases the host and strips a leading www prefix
3. The path gets a leading slash, loses a trailing slash, and is lowercased
4. Tracking utm params are filtered out and the rest sorted into the query
5. setdefault clusters every original URL under its full canonical form

Try it:

1. Add default-port stripping to the same function
2. Make path-lowercasing optional since real paths are case-sensitive
3. Print the chosen representative URL for each cluster
4. Add `https://EXAMPLE.com/page#top` and confirm it joins the page cluster.

⚠ Common Mistakes

- Lowercasing the path is fine only if the site treats paths case-insensitively; many do not.
- Filtering `kv` for emptiness is needed, or a query like `?` splits into a stray empty param.
- Applying the rules in the wrong order (sorting before stripping utm_) leaves the tracking param in the key.

Skills gained multi-rule canon url clustering rule composition

→ **Where You'll Use This:** Production deduplication pipelines chain exactly these rules so that all spellings of one page resolve to a single index entry, the linchpin of canonicalisation across an entire site.

⚡ Mental Model

Like running every dialect of an address through one standardiser at the post office, so a dozen scrawled envelopes all sort into the same mailbox.

Project 247 Keep a Content Fingerprint Stable

💡 The Idea Behind It

A good content fingerprint must ignore cosmetic edits like case and spacing yet still react to real changes in wording. This project normalises text by lowercasing and collapsing whitespace before hashing with SHA-1, so three differently-formatted versions of 'Hello World' produce one identical digest while adding an exclamation mark shifts it. The verdict line proves the first three fingerprints are equal, showing stability where it belongs.

What You'll Build & Learn

Show a fingerprint stays constant across cosmetic edits but changes on real ones.

- Normalise before hashing
- Stability against case and spacing
- Sensitivity to real edits

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 246: cluster duplicate urls to one canonical. Project 248 changes the surface problem while preserving the underlying method.

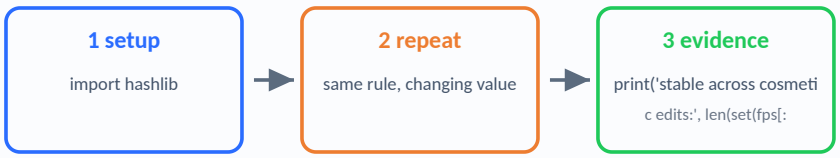
progress check

Reading target: keep a content fingerprint stable is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import hashlib` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('stable across cosmetic edits:', len(set(fps[:3])) == 1)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import hashlib`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`fps = [fingerprint(v) for v in versions]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('stable across cosmetic edits:', len(set(fps[:3])) == 1)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `fps = [fingerprint(v) for v in versions]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Keep a Content Fingerprint Stable: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: hashlib, fingerprint, text, sha1, join. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import hashlib | Core: fps = [fingerprint(v) for v in versions] | Evidence: print('stable across cosmetic edits:', len(set(fps[:3])) == 1)

🛠️ Reader to Writer Bridge

1. Read: explain keep a content fingerprint stable in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 246.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 248 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('stable across cosmetic edits:', len(set(fps[:3])) == 1)`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `import hashlib` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of keep a content fingerprint stable and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Lowercasing and collapsing whitespace before `hashlib.sha1` makes the first three variants hash identically, while real punctuation (`!`) still shifts the digest.

```
import hashlib

def fingerprint(text):
    return hashlib.sha1(' '.join(text.lower().split()).encode()).hexdigest()

versions = [
    'Hello World',
    'hello   world',
    '  HELLO WORLD  ',
    'Hello World!',
]
fps = [fingerprint(v) for v in versions]
for v, f in zip(versions, fps):
    print(f'{f[:10]} <- {v!r}')
print('stable across cosmetic edits:', len(set(fps[:3])) == 1)
```

```
output
2aae6c35c9 <- 'Hello World'
2aae6c35c9 <- 'hello   world'
2aae6c35c9 <- '  HELLO WORLD  '
430ce34d02 <- 'Hello World!'
stable across cosmetic edits: True
```

How it works:

1. fingerprint lowercases the text, collapses whitespace, encodes and hashes with SHA-1
2. Four versions vary by case, spacing and a trailing exclamation mark
3. The first three normalise identically and share one digest
4. The exclamation-mark version produces a different hash
5. A final line confirms the three cosmetic variants share one fingerprint

Try it:

1. Strip punctuation before hashing so the exclamation no longer matters
2. Add HTML stripping and confirm tags don't change the digest
3. Compare digests across two crawls to detect a real edit

4. Add `''hello world''` with a tab and confirm it still matches.

⚠ Common Mistakes

- Including punctuation in the normalised text makes trivial edits change the fingerprint.
- Stripping too aggressively (removing all punctuation) would merge genuinely different pages.
- Comparing the first 10 hex chars for the verdict risks a collision; compare full digests in production.

Skills gained `hashlib.sha1` `normalize then hash` `stability check`

→ Where You'll Use This: Stable fingerprints let you store one digest per page and detect genuine content changes between crawls without re-comparing full text, the basis of change-monitoring SEO tools.

Mental Model

Like a signature that should look the same whether you sign in pen or pencil, but visibly differs the moment you change your actual name.

Project 248 Find Duplicate Meta Descriptions

The Idea Behind It

Reused meta descriptions waste the snippet that earns clicks and hint at templated, thin pages. This project groups URLs by their exact description string with setdefault, then prints any description shared by more than one page along with every offending URL. Three pages reusing the running-gear blurb collapse into one group, exposing the copies that should each get a unique, compelling description.

What You'll Build & Learn

Group pages by meta description and list any description reused across URLs.

- Why duplicate descriptions cost clicks
- Grouping pages by description
- Listing offending URLs

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 247: keep a content fingerprint stable. Project 249 changes the surface problem while preserving the underlying method.

progress check

Reading target: find duplicate meta descriptions is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(' ', u)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(urls) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' ', u)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(urls) > 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Duplicate Meta Descriptions: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, https, example, com, Shop. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = { | Core: if len(urls) > 1: | Evidence: print(' ', u)

🛠️ Reader to Writer Bridge

1. Read: explain find duplicate meta descriptions in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 247.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 249 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' ', u)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of find duplicate meta descriptions and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Grouping URLs by their meta description and reporting groups larger than one finds every set of pages sharing a snippet, the descriptions to rewrite uniquely.

```
pages = {
    'https://example.com/a': 'Shop the best deals on running gear today.',
    'https://example.com/b': 'Shop the best deals on running gear today.',
    'https://example.com/c': 'Learn to repair your own bicycle at home.',
    'https://example.com/d': 'Shop the best deals on running gear today.',
}
groups = {}
for url, meta in pages.items():
    groups.setdefault(meta, []).append(url)
for meta in sorted(groups):
    urls = sorted(groups[meta])
    if len(urls) > 1:
        print(f'DUP META ({len(urls)}x): {meta!r}')
        for u in urls:
            print('  ', u)
```

```
output
DUP META (3x): 'Shop the best deals on running gear today.'
   https://example.com/a
   https://example.com/b
   https://example.com/d
```

How it works:

1. A dictionary maps four URLs to their meta description strings
2. setdefault appends each URL to a list keyed by its description
3. Descriptions are sorted for stable output
4. Any description with more than one URL prints a DUP META line with the count
5. Each offending URL is then printed indented beneath it

Try it:

1. Normalise descriptions by case and trimming before grouping
2. Flag descriptions over the snippet length limit as well
3. Count how many pages have no description at all
4. Lowercase and strip each description before grouping.

⚠ Common Mistakes

- Not normalising case or trailing whitespace splits descriptions that are effectively identical.
- Reporting single-URL groups buries the real duplicates in noise.
- Sorting only the groups but not the URLs inside makes the inner order nondeterministic.

Skills gained setdefault grouping | duplicate detection | meta audit

→ Where You'll Use This: Duplicate meta-description reports are routine in on-page audits; identical snippets across product pages reveal CMS templating and a missed chance to differentiate results pages.

Mental Model

Like several job applicants submitting the identical cover letter; the recruiter sees the copy-paste instantly and trusts each one less.

Project 249 Catch a Canonical Pointing at a Noindex Page

The Idea Behind It

Canonicalising to a page that is itself marked noindex sends opposite orders: 'index this representative' against 'do not index it'. This project stores each page with a canonical target and a noindex flag, then for every page looks up whether its canonical destination is noindexed. When it is, the line is flagged BAD because the engine may drop the entire cluster rather than index a forbidden representative.

What You'll Build & Learn

Detect pages whose canonical target carries a noindex directive.

- The canonical-versus-noindex conflict
- Looking up a target's flags
- Why the cluster can vanish

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 248: find duplicate meta descriptions. Project 250 changes the surface problem while preserving the underlying method.

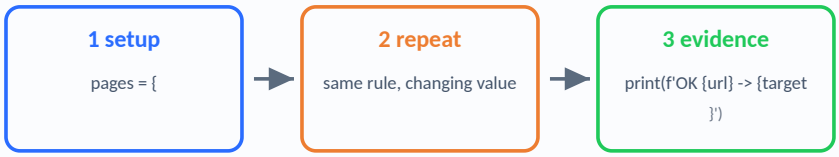
progress check

Reading target: catch a canonical pointing at a noindex page is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'OK {url} -> {target}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if pages[target]['noindex']:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'OK {url} -> {target}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if pages[target]['noindex']:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Catch a Canonical Pointing at a Noindex Page: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, https, example, com, canonical. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: if pages[target]['noindex']: | Evidence: print(f'OK {url} -> {target}')

🛠️ Reader to Writer Bridge

1. Read: explain catch a canonical pointing at a noindex page in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 248.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 250 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'OK {url} -> {target}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of catch a canonical pointing at a noindex page and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Looking up the canonical target's own noindex flag reveals the conflict: a representative URL chosen for indexing must not itself be excluded from the index.

```
pages = {
    'https://example.com/a': {'canonical': 'https://example.com/b', 'noindex':
False},
    'https://example.com/b': {'canonical': 'https://example.com/b', 'noindex':
True},
    'https://example.com/c': {'canonical': 'https://example.com/c', 'noindex':
False},
}
for url in sorted(pages):
    target = pages[url]['canonical']
    if pages[target]['noindex']:
        print(f'BAD {url} -> canonical {target} is noindex (signal conflict)')
    else:
        print(f'OK  {url} -> {target}')
```

```
output
BAD https://example.com/a -> canonical https://example.com/b is noindex (signal
conflict)
BAD https://example.com/b -> canonical https://example.com/b is noindex (signal
conflict)
OK  https://example.com/c -> https://example.com/c
```

How it works:

1. Each page maps to a dict with a canonical URL and a noindex boolean
2. The loop reads each page's canonical target
3. It indexes back into pages to check that target's noindex flag
4. If the target is noindex it prints a BAD signal-conflict line
5. Otherwise it prints an OK line with the resolved target

Try it:

1. Also check the target returns a 200 status, not just indexable
2. Report the orphaned pages that lose their only representative
3. Suggest the nearest indexable alternative canonical
4. Clear page b's noindex flag and watch both verdicts turn OK.

⚠ Common Mistakes

- Checking the source page's noindex instead of the canonical target's misses the conflict.
- A KeyError occurs if a canonical names a target not present in the page set.
- Treating a self-noindex page as fine ignores that it still poisons pages pointing at it.

Skills gained `dict lookup` `boolean flag` `signal conflict`

→ **Where You'll Use This:** This conflict silently de-indexes pages in real audits; checking that every canonical target is itself indexable prevents accidentally pointing a whole duplicate group at a forbidden URL.

⚡ Mental Model

Like naming an heir in your will who has already renounced the inheritance; the estate has no valid recipient and the whole bequest collapses.

Project 250 Verify a Canonical Resolves to a 200

💡 The Idea Behind It

A canonical that points at a 404 or a redirect is wasted because the engine cannot index a representative that fails to return a live page. This project keeps a status-code table and a canonical map, then for each page looks up its canonical's HTTP code. Only a 200 passes; a 404 or a 301 target prints a BAD line explaining the canonical must resolve to a live, final page.

What You'll Build & Learn

Verify each canonical target returns a 200 rather than a 404 or redirect.

- Canonicals must resolve to 200
- Looking up a target's status code
- Why redirects and 404s fail

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 249: catch a canonical pointing at a noindex page. Project 251 changes the surface problem while preserving the underlying method.

progress check

Reading target: verify a canonical resolves to a 200 is the surface task; the transferable Python move is to mark setup, rule, and output before running.

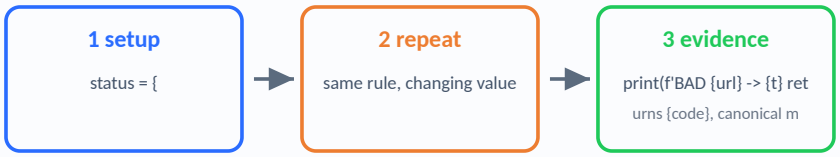
setup -> rule -> result

🔎 Logic Lens

- Setup: `status = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'BAD {url} -> {t} returns {code}, canonical must be a 200 page')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`status = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if code == 200:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'BAD {url} -> {t} returns {code}, canonical must be a 200 page')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if code == 200:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Verify a Canonical Resolves to a 200: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: status, https, example, com, canonical. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: status = { | Core: if code == 200: | Evidence: print(f'BAD {url} -> {t} returns {code}, canonical must be a 200 page')

🛠️ Reader to Writer Bridge

1. Read: explain verify a canonical resolves to a 200 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 249.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 251 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'BAD {url} -> {t} returns {code}, canonical must be a 200 page')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `status = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of verify a canonical resolves to a 200 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Looking up the canonical target's status code and requiring exactly 200 catches canonicals aimed at missing or redirecting URLs, which silently break the indexing signal.

```
status = {
    'https://example.com/a': 200,
    'https://example.com/b': 404,
    'https://example.com/c': 301,
    'https://example.com/d': 200,
}
canonical = {
    'https://example.com/a': 'https://example.com/b',
    'https://example.com/d': 'https://example.com/c',
    'https://example.com/c': 'https://example.com/c',
}
for url in sorted(canonical):
    t = canonical[url]
    code = status.get(t, 0)
    if code == 200:
        print(f'OK  {url} -> {t} (200)')
    else:
        print(f'BAD {url} -> {t} returns {code}, canonical must be a 200 page')
```

```
output
BAD https://example.com/a -> https://example.com/b returns 404, canonical must be a
200 page
BAD https://example.com/c -> https://example.com/c returns 301, canonical must be a
200 page
BAD https://example.com/d -> https://example.com/c returns 301, canonical must be a
200 page
```

How it works:

1. A status dict maps URLs to HTTP codes including 200, 404 and 301
2. A canonical dict maps pages to their declared canonical targets
3. The loop reads each target and fetches its code from status with a default of zero
4. A target returning 200 prints OK, anything else prints BAD with the code

Try it:

1. Resolve a 301 target one hop further and re-test its final code

2. Treat any 3xx as a redirect rather than only 301
3. Tally how many canonicals are broken across the set
4. Point page c's canonical at page a and watch its verdict turn OK.

⚠ Common Mistakes

- Accepting any 3xx as fine ignores that a canonical should name the final destination, not a redirect.
- Using `status[t]` instead of `.get` raises if the target is not in the status map.
- Checking the source URL's status instead of the canonical target's misses the problem entirely.

Skills gained dict.get default | status validation | canonical audit

→ Where You'll Use This: Validating that canonical targets answer 200 is a standard crawl check; pointing at broken or redirecting URLs strands ranking signals and is a frequent post-migration defect.

Mental Model

Naming a representative who's moved away or doesn't exist: the engine knocks on the canonical's door and finds nobody home.

Project 251 Flag a Cross-Domain Canonical

The Idea Behind It

A canonical pointing to a different registrable domain hands indexing credit to another site, which is rarely intended outside syndication. This project extracts the host from each canonical URL and tests whether it equals the configured site or ends with a dot plus that site, marking same-site or CROSS-DOMAIN. A subdomain of example.com counts as same-site, while other-site.com is correctly flagged as foreign.

What You'll Build & Learn

Flag canonicals that point outside the configured registrable domain.

- Cross-domain canonicals leak authority
- Extracting a host from a URL
- Matching domain and its subdomains

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 250: verify a canonical resolves to a 200. Project 252 changes the surface problem while preserving the underlying method.

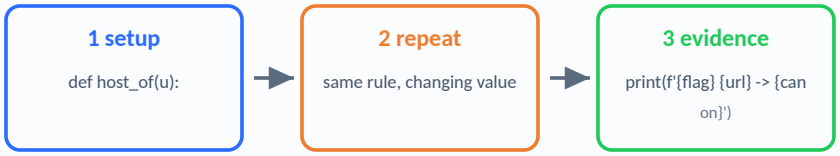
progress check

Reading target: flag a cross-domain canonical is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `def host_of(u):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'{flag} {url} -> {canon}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def host_of(u):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`flag = 'same-site' if same else 'CROSS-DOMAIN'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{flag} {url} -> {canon}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = 'same-site' if same else 'CROSS-DOMAIN'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Flag a Cross-Domain Canonical: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: host_of, split, lower, pages, https. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def host_of(u): | Core: flag = 'same-site' if same else 'CROSS-DOMAIN' | Evidence: print(f'{flag} {url} -> {canon}')

🛠️ Reader to Writer Bridge

1. Read: explain flag a cross-domain canonical in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 250.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 252 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{flag} {url} -> {canon}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def host_of(u):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of flag a cross-domain canonical and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Extracting the canonical target's host and accepting it only when it equals the site or ends with `.site` treats subdomains as same-site while flagging a foreign domain.

```
def host_of(u):
    return u.split('://', 1)[1].split('/', 1)[0].lower()

pages = {
    'https://blog.example.com/post': 'https://www.example.com/post',
    'https://shop.example.com/item': 'https://shop.example.com/item',
    'https://news.example.com/x': 'https://other-site.com/x',
}
site = 'example.com'
for url in sorted(pages):
    canon = pages[url]
    ch = host_of(canon)
    same = ch == site or ch.endswith('.' + site)
    flag = 'same-site' if same else 'CROSS-DOMAIN'
    print(f'{flag} {url} -> {canon}')
```

```
output
same-site https://blog.example.com/post -> https://www.example.com/post
CROSS-DOMAIN https://news.example.com/x -> https://other-site.com/x
same-site https://shop.example.com/item -> https://shop.example.com/item
```

How it works:

1. host_of pulls the host out of a URL by splitting on the scheme then the first slash
2. Each page maps to its canonical, and the site domain is configured as example.com
3. The check passes when the canonical host equals the site or ends with dot-site
4. Same-site canonicals are labelled, foreign ones flagged CROSS-DOMAIN

Try it:

1. Handle URLs with ports or userinfo in the host portion
2. Maintain an allowlist of approved syndication partners
3. Strip a leading www before comparing hosts
4. Add a page canonicalising to `https://evil-example.com/x` and confirm it is CROSS-DOMAIN.

⚠ Common Mistakes

- Using `in` instead of `endswith` would match `notexample.com` as same-site.

- Comparing full URLs rather than hosts misclassifies same-host pages with different paths.
- Forgetting to lowercase the host lets a capitalised domain slip through as cross-domain.

Skills gained `str.split` `domain matching` `cross-domain check`

→ Where You'll Use This: Cross-domain canonical audits catch accidental authority leaks and unauthorised syndication; deliberate cross-domain canonicals exist but should be explicit and verified, not silent surprises.

Mental Model

Like signing your house over to a stranger's name on the deed: unless you meant to gift it, the equity quietly leaves your family.

Project 252 Pick the Representative URL for a Duplicate Set

The Idea Behind It

When many URLs serve one page you must pick a single canonical, and a common deterministic policy prefers the shortest URL with alphabetical tie-breaking. This project applies min over the duplicate list with a key of length-then-string, choosing the cleanest address, then lists every other variant as a fold target. The bare product URL wins over its slashed and parameter-laden siblings.

What You'll Build & Learn

Choose one representative URL from a duplicate set by a shortest-then-alphabetical rule.

- Choosing a representative URL
- A shortest-then-alphabetical policy
- Determinism via tie-breaking

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 251: flag a cross-domain canonical. Project 253 changes the surface problem while preserving the underlying method.

progress check

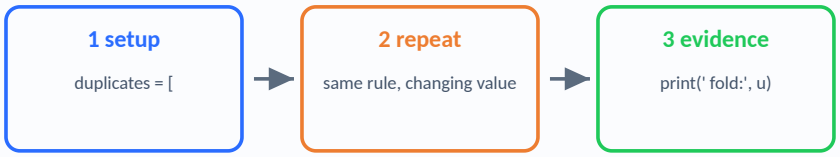
Reading target: pick the representative url for a duplicate set is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `duplicates = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(' fold:', u)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`duplicates = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if u != chosen:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' fold:', u)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if u != chosen:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Pick the Representative URL for a Duplicate Set: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: duplicates, https, example, com, product. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: duplicates = [| Core: if u != chosen: | Evidence: print(' fold:', u)

🛠️ Reader to Writer Bridge

1. Read: explain pick the representative url for a duplicate set in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 251.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 253 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' fold:', u)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `duplicates = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of pick the representative url for a duplicate set and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A sort key of `(len(u), u)` makes `min` prefer the shortest URL and fall back to alphabetical order, yielding a deterministic representative that the rest fold into.

```
duplicates = [
    'https://example.com/product?ref=ad&color=red',
    'https://example.com/product',
    'https://example.com/product/',
    'https://example.com/product?color=red',
]
chosen = min(duplicates, key=lambda u: (len(u), u))
print('canonical:', chosen)
for u in sorted(duplicates):
    if u != chosen:
        print('  fold:', u)
```

```
output
canonical: https://example.com/product
  fold: https://example.com/product/
  fold: https://example.com/product?color=red
  fold: https://example.com/product?ref=ad&color=red
```

How it works:

1. A list holds four spellings of one product page
2. min picks the URL minimising the tuple of length then the string itself
3. The length component favours the shortest, cleanest address
4. The string component breaks ties deterministically
5. Every other sorted URL is printed as a fold candidate

Try it:

1. Prefer URLs without query parameters over short ones
2. Add a rule favouring https over http variants
3. Let the policy prefer a manually-pinned canonical when present
4. Change the policy to prefer the URL with the fewest query params.

⚠ Common Mistakes

- Using `min` on length alone leaves ties resolved by list order, which is nondeterministic across crawls.
- Picking the longest URL canonicalises to the param-laden variant, the opposite of what you want.
- Not excluding the chosen URL from the fold list reports the canonical as folding into itself.

Skills gained min with key | tuple sort key | canonical selection

→ Where You'll Use This: Choosing the canonical representative is the decision step of any dedupe pipeline; a deterministic rule keeps the index stable across recrawls and avoids flip-flopping the chosen URL.

Mental Model

Picking the captain of identical twins by a fixed rule, like whoever's name comes first, so you always pick the same one every time.

Project 253 Measure Index Bloat from Canonicals

The Idea Behind It

Index bloat is the gap between how many URLs are crawlable and how many distinct pages they truly represent. This project maps each crawlable URL to its canonical, counts the total URLs and the unique canonical targets via a set, and computes the duplicate share as a percentage. Five URLs resolving to two real pages reveal sixty percent bloat, quantifying the crawl budget wasted on redundant addresses.

What You'll Build & Learn

Quantify the duplicate share of a crawlable set as an index-bloat percentage.

- Defining index bloat
- Counting unique canonicals with a set
- Computing a duplicate percentage

Skill Roadmap

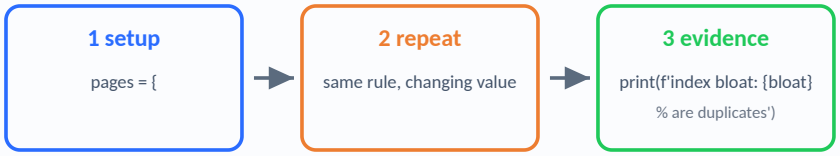
Skill step: pushes the same idea into a less automatic situation. It builds on Project 252: pick the representative url for a duplicate set. Project 254 changes the surface problem while preserving the underlying method. *progress check*

Reading target: measure index bloat from canonicals is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'index bloat: {bloat}% are duplicates')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`canonicals = {p['canonical'] if isinstance(p, dict) else p for p in pages.values()}`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'index bloat: {bloat}% are duplicates')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `canonicals = {p['canonical'] if isinstance(p, dict) else p for p in pages.values()}`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure Index Bloat from Canonicals: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, https, example, com, total. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = { | Core: canonicals = {p['canonical'] if isinstance(p, dict) else p for p in pag... | Evidence: print(f'index bloat: {bloat}% are duplicates')

Reader to Writer Bridge

1. Read: explain measure index bloat from canonicals in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 252.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 254 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'index bloat: {bloat}% are duplicates')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure index bloat from canonicals and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

From Concept to Code (Python)

The set of distinct canonical targets is the true page count; the percentage of crawlable URLs beyond that count quantifies how much of the site is duplicate.

```
pages = {
    'https://example.com/a': 'https://example.com/a',
    'https://example.com/b': 'https://example.com/a',
    'https://example.com/c': 'https://example.com/c',
```

```
    'https://example.com/d': 'https://example.com/c',
    'https://example.com/e': 'https://example.com/c',
}
total = len(pages)
canonicals = {p['canonical'] if isinstance(p, dict) else p for p in
pages.values()}
unique = len(canonicals)
bloat = round((total - unique) / total * 100, 1)
print(f'crawlable URLs: {total}')
print(f'unique canonical pages: {unique}')
print(f'index bloat: {bloat}% are duplicates')
```

```
output
crawlable URLs: 5
unique canonical pages: 2
index bloat: 60.0% are duplicates
```

How it works:

1. A dict maps five crawlable URLs to their canonical targets
2. total is the count of all crawlable URLs
3. A set comprehension over the canonical values yields the unique page count
4. bloat is the difference over total as a rounded percentage
5. It prints crawlable count, unique pages and the bloat percentage

Try it:

1. Break the bloat figure down per directory or template
2. Handle canonical values stored as plain strings or dicts uniformly
3. Report the absolute number of wasted URLs alongside the percentage
4. Make every canonical self-referencing and watch bloat drop to 0%.

⚠ Common Mistakes

- Counting raw URLs as indexed pages massively over-reports the real index size.
- Dividing by `unique` instead of `total` measures a different and confusing ratio.
- Forgetting to dedupe canonicals into a set leaves duplicate targets in the count.

Skills gained set dedupe | percentage math | bloat metric

→ Where You'll Use This: Index-bloat metrics justify cleanup work to stakeholders; showing that most crawlable URLs are duplicates makes the case for canonical tags, redirects and parameter handling in concrete numbers.

Mental Model

Like a library catalogue listing five call numbers that all shelve to two actual books: the catalogue looks bigger than the collection really is.

Project 254 Detect Soft-404 Thin Pages

The Idea Behind It

A soft 404 returns a 200 status but carries no real content behind phrases like 'no results found' or 'page not available'. This project combines two weak signals into one strong one: it counts word tokens with a regex and scans the lowercased text for known error markers, flagging a page as SOFT-404 only when it is both thin and matches a marker. A genuine policy page with twenty-four words passes.

What You'll Build & Learn

Catch soft-404 pages by pairing a thin word count with error-phrase markers.

- What a soft 404 is
- Combining two weak signals
- Matching error phrasing

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 253: measure index bloat from canonicals. Project 255 changes the surface problem while preserving the underlying method.

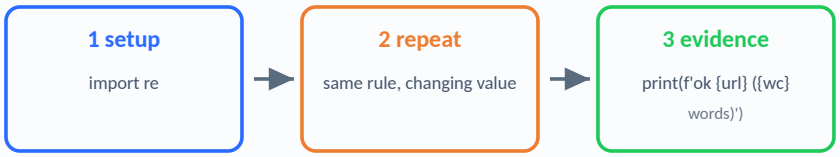
progress check

Reading target: detect soft-404 thin pages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'ok {url} ({wc} words)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`hit = next((m for m in soft404_markers if m in text), None)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'ok {url} ({wc} words)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `hit = next((m for m in soft404_markers if m in text), None)`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Detect Soft-404 Thin Pages: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: re, pages, https, example, com. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: hit = next((m for m in soft404_markers if m in text), None) | Evidence: print(f'ok {url} ({wc} words)')

🛠️ Reader to Writer Bridge

1. Read: explain detect soft-404 thin pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 253.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 255 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'ok {url} ({wc} words)')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect soft-404 thin pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Requiring both a low word count AND an error phrase avoids flagging a short-but-real page, catching the 200-status pages that behave like a 404.

```
import re
pages = {
    'https://example.com/real': 'Our return policy lasts thirty days from the
delivery date. Items must be unused and in original packaging to qualify for a
full refund.',
    'https://example.com/empty': 'Sorry, no results were found for your search.
```

```
Try again.',
    'https://example.com/error': 'Page not available. Please go back to the
homepage.',
}
soft404_markers = ('no results', 'not available', 'go back', 'sorry')
for url in sorted(pages):
    text = pages[url].lower()
    wc = len(re.findall(r"[a-z']+", text))
    hit = next((m for m in soft404_markers if m in text), None)
    if wc < 15 and hit:
        print(f'SOFT-404 {url} ({wc} words, matched {hit!r})')
    else:
        print(f'ok       {url} ({wc} words)')
```

```
output
SOFT-404 https://example.com/empty (10 words, matched 'no results')
SOFT-404 https://example.com/error (9 words, matched 'not available')
ok       https://example.com/real (24 words)
```

How it works:

1. A dict maps three URLs to bodies ranging from real content to error messages
2. A tuple lists soft-404 marker phrases like no results and not available
3. re.findall counts the word tokens in each lowercased body
4. next finds the first marker phrase present in the text, or None
5. A page under fifteen words and matching a marker is flagged SOFT-404

Try it:

1. Add more marker phrases such as nothing here and try later
2. Require the marker near the top of the text, not just anywhere
3. Tune the word threshold and re-evaluate borderline pages
4. Lower the word threshold to 8 and watch the error page clear.

⚠ Common Mistakes

- Flagging on the error phrase alone would catch a long article that quotes 'no results'.
- Flagging on word count alone trips on legitimately brief pages.
- Case-sensitive matching misses 'No Results' if you do not lowercase first.

Skills gained re.findall | phrase matching | combined heuristic

→ Where You'll Use This: Soft-404 detection matters because these pages waste crawl budget and pollute the index with empty search-result and error templates that return 200 instead of a proper 404.

Mental Model

Like a shop with the lights on and the open sign lit but empty shelves inside: the status says welcome, the content says there's nothing here.

Project 255 Compare Pages with Character Shingles

The Idea Behind It

Word shingles miss tiny spelling differences such as color versus colour, so this project compares at the character level instead. It strips spaces, lowercases the text, and builds a set of 4-character grams with a sliding window, then computes Jaccard over those character n-grams. The British and American spellings share most grams but the score of 0.5 still falls under the 0.6 cutoff, marking them distinct here.

What You'll Build & Learn

Compare two texts with character 4-grams and judge them by a Jaccard cutoff.

- Character n-grams versus word shingles
- Catching small spelling differences
- Applying a similarity cutoff

Skill Roadmap

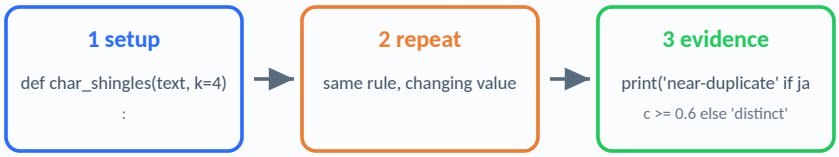
Skill step: pushes the same idea into a less automatic situation. It builds on Project 254: detect soft-404 thin pages. Project 256 changes the surface problem while preserving the underlying method. *progress check*

Reading target: compare pages with character shingles is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def char_shingles(text, k=4):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('near-duplicate' if jac >= 0.6 else 'distinct')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def char_shingles(text, k=4):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`print('near-duplicate' if jac >= 0.6 else 'distinct')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('near-duplicate' if jac >= 0.6 else 'distinct')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('near-duplicate' if jac >= 0.6 else 'distinct')`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Compare Pages with Character Shingles: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: char_shingles, text, lower, replace, range. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def char_shingles(text, k=4): | Core: print('near-duplicate' if jac >= 0.6 else 'distinct') | Evidence: print('near-duplicate' if jac >= 0.6 else 'distinct')

🛠️ Reader to Writer Bridge

1. Read: explain compare pages with character shingles in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 254.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 256 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('near-duplicate' if jac >= 0.6 else 'distinct')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `def char_shingles(text, k=4):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compare pages with character shingles and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Sliding a k-character window over the spaces-removed text builds character n-grams, and their Jaccard similarity reacts to letter-level edits that word shingles would round away.

```
def char_shingles(text, k=4):
    t = text.lower().replace(' ', '')
    return {t[i:i + k] for i in range(len(t) - k + 1)}

a = char_shingles('color of the sky')
```

```
b = char_shingles('colour of the sky')
jac = round(len(a & b) / len(a | b), 3)
print('char 4-grams A:', len(a))
print('char 4-grams B:', len(b))
print('jaccard:', jac)
print('near-duplicate' if jac >= 0.6 else 'distinct')
```

```
output
char 4-grams A: 10
char 4-grams B: 11
jaccard: 0.5
distinct
```

How it works:

1. char_shingles lowercases the text and removes spaces
2. A set comprehension forms every 4-character window across the string
3. Two near-identical spellings become gram sets a and b
4. Jaccard divides their intersection by their union, rounded
5. A score at or above 0.6 prints near-duplicate, otherwise distinct

Try it:

1. Lower k to three and see the similarity rise
2. Keep spaces in the grams and compare the effect
3. Adjust the 0.6 cutoff until these two count as duplicates
4. Lower the threshold to 0.5 and watch the verdict flip to near-duplicate.

⚠ Common Mistakes

- Leaving spaces in lets word boundaries dominate the n-grams and dilute the comparison.
- Picking k too small (1 or 2) makes almost everything similar.
- Forgetting that `set` dedupes repeated n-grams, which can deflate counts on repetitive text.

Skills gained char n-grams | set jaccard | cutoff verdict

→ Where You'll Use This: Character n-grams catch typo, localisation and inflection differences that word-level methods overlook, useful for spotting near-identical product copy that varies only in spelling across regional sites.

Mental Model

Reading by spelling out letters rather than whole words, so 'colour' and 'color' look nearly the same even though the word-level scan called them different.

Project 256 Cosine Similarity of Two Pages

The Idea Behind It

Cosine similarity treats each page as a vector of word counts and measures the angle between them, so it ignores total length and rewards shared vocabulary even when terms repeat. This project builds a Counter for each text, then computes the dot product over shared words divided by the product of the vector magnitudes. Two phrases overlapping heavily but differing in repetition score a high 0.849.

What You'll Build & Learn

Compute the cosine similarity of two pages from their word-count vectors.

- Pages as word-count vectors
- The cosine angle measure
- Why repetition is tolerated

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 255: compare pages with character shingles. Project 257 changes the surface problem while preserving the underlying method.

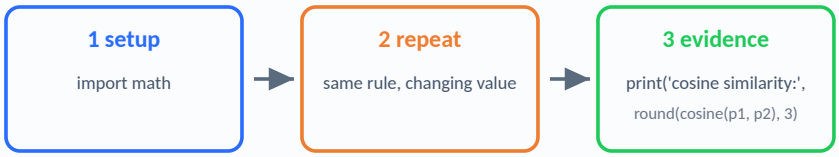
progress check

Reading target: cosine similarity of two pages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import math` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('cosine similarity:', round(cosine(p1, p2), 3))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import math`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`dot = sum(a[t] * b[t] for t in common)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('cosine similarity:', round(cosine(p1, p2), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dot = sum(a[t] * b[t] for t in common)`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Cosine Similarity of Two Pages: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: math, collections, Counter, vec, text. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import math | Core: dot = sum(a[t] * b[t] for t in common) | Evidence: print('cosine similarity:', round(cosine(p1, p2), 3))

🛠️ Reader to Writer Bridge

1. Read: explain cosine similarity of two pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 255.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 257 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('cosine similarity:', round(cosine(p1, p2), 3))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import math` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of cosine similarity of two pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The dot product over shared terms divided by the product of the vector magnitudes gives the cosine of the angle between two term-frequency vectors, a length-insensitive similarity.

```
import math
from collections import Counter

def vec(text):
    return Counter(text.lower().split())
```

```
def cosine(a, b):
    common = set(a) & set(b)
    dot = sum(a[t] * b[t] for t in common)
    na = math.sqrt(sum(v * v for v in a.values()))
    nb = math.sqrt(sum(v * v for v in b.values()))
    return dot / (na * nb)

p1 = vec('buy cheap shoes cheap shoes online')
p2 = vec('buy cheap shoes online today')
print('cosine similarity:', round(cosine(p1, p2), 3))
```

```
output
cosine similarity: 0.849
```

How it works:

1. vec builds a Counter mapping each word to its frequency
2. cosine finds the words common to both vectors
3. The dot product sums the products of counts over those shared words
4. Each vector's magnitude is the square root of its summed squares
5. Dividing the dot product by the product of magnitudes gives the cosine score

Try it:

1. Remove stop words before vectorising for sharper scores
2. Use TF-IDF weights instead of raw counts
3. Guard against an empty vector that would divide by zero
4. Make the two pages identical and confirm cosine is 1.0.

⚠ Common Mistakes

- Summing over all terms instead of the shared set wastes work; absent terms contribute zero anyway.
- Dividing by zero when a page is empty crashes; guard against an empty vector.
- Confusing cosine with Jaccard: cosine weights term frequency, Jaccard only set membership.

Skills gained collections.Counter | dot product | vector cosine

→ Where You'll Use This: Cosine similarity is the workhorse of text comparison in many SEO and search systems, scoring topical overlap between pages independent of how long or repetitive each one is.

Mental Model

Two arrows shot from the same origin; cosine measures the angle between them, caring about direction of vocabulary rather than how far each arrow flies.

Project 257 Cluster Exact Duplicates with SHA-1

The Idea Behind It

To dedupe a whole site by exact content, hash each page body and group identical digests together. This project computes a SHA-1 over each raw body, buckets URLs under their digest with setdefault, then sorts the clusters by size. For any cluster larger than one it keeps the alphabetically first URL and lists the rest as drops, turning byte-identical pages into a clear keep-and-remove plan.

What You'll Build & Learn

Cluster byte-identical page bodies and choose one keeper per cluster.

- Clustering exact duplicates by hash
- Sorting clusters by size
- Choosing a keeper and drops

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 256: cosine similarity of two pages. Project 258 changes the surface problem while preserving the underlying method.

progress check

Reading target: cluster exact duplicates with sha-1 is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

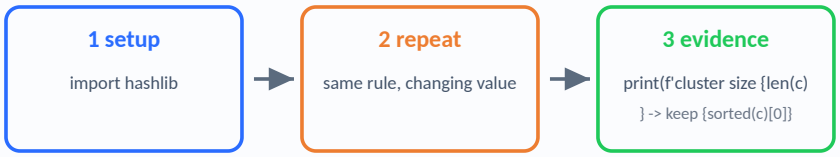
- Setup: `import hashlib` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'cluster size {len(c)} -> keep {sorted(c)[0]}, drop {sorted(c)[1:]}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import hashlib`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(c) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'cluster size {len(c)} -> keep {sorted(c)[0]}, drop {sorted(c)[1:]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(c) > 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Cluster Exact Duplicates with SHA-1: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: hashlib, pages, https, example, com. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import hashlib | Core: if len(c) > 1: | Evidence: print(f'cluster size {len(c)} -> keep {sorted(c)[0]}, drop {sorted(c)[1...

🛠️ Reader to Writer Bridge

1. Read: explain cluster exact duplicates with sha-1 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 256.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 258 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'cluster size {len(c)} -> keep {sorted(c)[0]}, drop {sorted(c)[1:]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import hashlib` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of cluster exact duplicates with sha-1 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Grouping URLs by `hashlib.sha1(body).hexdigest()` clusters byte-identical pages; sorting each cluster makes the keeper deterministic and lists the rest to fold away.

```
import hashlib
pages = {
    'https://example.com/a': 'identical body text here',
    'https://example.com/b': 'identical body text here',
    'https://example.com/c': 'a totally different article body',
    'https://example.com/d': 'identical body text here',
```

```
}
groups = {}
for url, body in pages.items():
    h = hashlib.sha1(body.encode()).hexdigest()
    groups.setdefault(h, []).append(url)
clusters = sorted(groups.values(), key=len, reverse=True)
for c in clusters:
    if len(c) > 1:
        print(f'cluster size {len(c)} -> keep {sorted(c)[0]}, drop {sorted(c)
[1:]}')
```

```
output
cluster size 3 -> keep https://example.com/a, drop ['https://example.com/b',
'https://example.com/d']
```

How it works:

1. A dict maps four URLs to body strings, three of them identical
2. Each body is hashed with SHA-1 and the URL appended under that digest
3. Clusters are the grouped value lists, sorted by length descending
4. For any cluster bigger than one, the sorted-first URL is the keeper
5. The remaining URLs in the cluster are listed as drops

Try it:

1. Normalise whitespace and case before hashing for looser matching
2. Pick the keeper by shortest URL instead of alphabetical
3. Emit redirect rules from each dropped URL to its keeper
4. Normalise whitespace and case before hashing to catch cosmetic copies.

⚠ Common Mistakes

- Not sorting the cluster makes which URL is kept depend on dict insertion order.
- Hashing without normalising means a single extra space splits a true duplicate.
- Reporting singleton clusters clutters the output with unique pages.

Skills gained hashlib.sha1 | setdefault grouping | cluster keeper

→ Where You'll Use This: Exact-duplicate clustering produces the actionable output of a dedupe pass: a list of which copy to keep and which URLs to redirect or canonicalise into it across the whole site.

⚡ Mental Model

Sorting a box of photocopies into stacks of identical sheets, then keeping the top sheet of each tall stack and recycling the rest.

Project 258 Fold Mixed-Case Host Duplicates

The Idea Behind It

Hostnames are case-insensitive so Example.com and example.com hit the same server, but URL paths are case-sensitive and must be left untouched. This project lowercases only the host portion while preserving the path exactly, then groups originals under that canonical host. Three case spellings of the same path collapse together, while a differently-cased path would correctly stay separate because the path is not folded.

What You'll Build & Learn

Fold URLs that differ only in host letter-case while keeping paths intact.

- Hosts are case-insensitive
- Paths are case-sensitive
- Lowercasing only the host

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 257: cluster exact duplicates with sha-1. Project 259 changes the surface problem while preserving the underlying method.

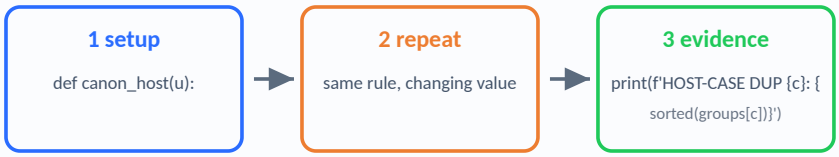
progress check

Reading target: fold mixed-case host duplicates is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def canon_host(u):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'HOST-CASE DUP {c}: {sorted(groups[c])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def canon_host(u):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(groups[c]) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'HOST-CASE DUP {c}: {sorted(groups[c])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(groups[c]) > 1:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Fold Mixed-Case Host Duplicates: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: canon_host, scheme, rest, split, host. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: def canon_host(u): | Core: if len(groups[c]) > 1: | Evidence: print(f'HOST-CASE DUP {c}: {sorted(groups[c])}')

🛠️ Reader to Writer Bridge

1. Read: explain fold mixed-case host duplicates in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 257.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 259 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'HOST-CASE DUP {c}: {sorted(groups[c])}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def canon_host(u):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of fold mixed-case host duplicates and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Lowercasing the host while leaving the path untouched groups the three `/Page` host variants together but keeps `/Other` distinct, respecting the path's case sensitivity.

```
def canon_host(u):
    scheme, rest = u.split('://', 1)
    host, slash, path = rest.partition('/')
    return scheme + '://' + host.lower() + '/' + path
urls = [
    'https://Example.com/Page',
    'https://EXAMPLE.com/Page',
```

```
    'https://example.com/Page',
    'https://example.com/Other',
]
groups = {}
for u in urls:
    groups.setdefault(canon_host(u), []).append(u)
for c in sorted(groups):
    if len(groups[c]) > 1:
        print(f'HOST-CASE DUP {c}: {sorted(groups[c])}')
```

```
output
HOST-CASE DUP https://example.com/Page: ['https://EXAMPLE.com/Page',
'https://Example.com/Page', 'https://example.com/Page']
```

How it works:

1. canon_host splits scheme from the rest then partitions host from path
2. Only the host is lowercased while the path is rejoined unchanged
3. A list mixes several case spellings of one host and path
4. setdefault groups originals under each canonical host string
5. Multi-member groups print a HOST-CASE DUP line with all variants

Try it:

1. Add a differently-cased path and confirm it stays separate
2. Strip a leading www in the same step
3. Lowercase the path too and observe the incorrect merge
4. Add `https://example.com/page` (lowercase path) and confirm it stays separate.

⚠ Common Mistakes

- Lowercasing the whole URL would also merge `/Page` with a hypothetical `/page`.
- Using `split('/')` instead of `partition('/')` raises if a URL has no path.
- Assuming paths are case-insensitive; many servers treat them as distinct resources.

Skills gained str.partition | selective lowercasing | host folding

→ Where You'll Use This: Mixed-case host duplicates arise from inconsistent linking and CMS output; folding only the host respects the web's case rules and avoids wrongly merging genuinely distinct paths.

⚡ Mental Model

Like a street name that doesn't care about capitals but a house number that does: SMITH STREET equals Smith Street, but 12A is not 12a.

Project 259 Fold a Default-Port Duplicate URL

The Idea Behind It

Over HTTPS the default port is 443, so example.com:443 and plain example.com are the same origin reached two ways. This project detects a host ending in colon-443 and removes those four characters, rebuilding the URL without the redundant port, then groups originals under that key. The explicit-port login URL folds together with its bare-host twin while a single-port page stays on its own.

What You'll Build & Learn

Fold a URL carrying an explicit default 443 port with its portless twin.

- The default HTTPS port is 443
- Stripping an explicit default port
- Folding origin duplicates

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 258: fold mixed-case host duplicates. Project 260 changes the surface problem while preserving the underlying method.

progress check

Reading target: fold a default-port duplicate url is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

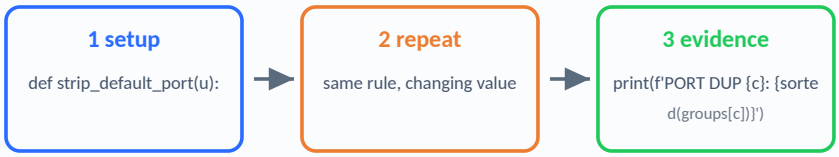
- Setup: `def strip_default_port(u):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'PORT DUP {c}: {sorted(groups[c])}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def strip_default_port(u):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if host.endswith(':443'):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'PORT DUP {c}: {sorted(groups[c])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if host.endswith(':443'):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Fold a Default-Port Duplicate URL: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: strip_default_port, scheme, rest, split, host. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: def strip_default_port(u): | Core: if host.endswith(':443'): | Evidence: print(f'PORT DUP {c}: {sorted(groups[c])}')

🛠️ Reader to Writer Bridge

1. Read: explain fold a default-port duplicate url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 258.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 260 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'PORT DUP {c}: {sorted(groups[c])}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `def strip_default_port(u):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of fold a default-port duplicate url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Removing a trailing `:443` from the host normalises the explicit default port to the implicit one, so the two `/login` URLs collapse into a single canonical origin.

```
def strip_default_port(u):
    scheme, rest = u.split('://', 1)
    host, slash, path = rest.partition('/')
    if host.endswith(':443'):
        host = host[:-4]
    return scheme + '://' + host + '/' + path
urls = [
```

```
    'https://example.com:443/login',
    'https://example.com/login',
    'https://example.com:443/home',
]
groups = {}
for u in urls:
    groups.setdefault(strip_default_port(u), []).append(u)
for c in sorted(groups):
    if len(groups[c]) > 1:
        print(f'PORT DUP {c}: {sorted(groups[c])}')
```

```
output
PORT DUP https://example.com/login: ['https://example.com/login',
'https://example.com:443/login']
```

How it works:

1. strip_default_port splits scheme from the rest and partitions host from path
2. If the host ends in colon-443 those four characters are sliced off
3. The URL is rebuilt with the cleaned host and original path
4. setdefault groups originals under the portless canonical
5. Groups with more than one variant print a PORT DUP line

Try it:

1. Also strip colon-80 when the scheme is http
2. Handle a host that carries a non-default port and leave it intact
3. Generalise to read the default port from the scheme
4. Add `https://example.com:8443/login` and confirm it stays separate.

⚠ Common Mistakes

- Stripping `:443` anywhere in the URL could damage a path segment that happens to contain it.
- Forgetting that a non-default port like `:8443` is a genuinely different origin and must stay.
- Slicing `host[:-4]` only works because `:443` is exactly four characters; a different port needs a split.

Skills gained str.endswith | slice trimming | port normalization

→ Where You'll Use This: Explicit default ports occasionally leak into links from proxies or misconfigurations; folding them prevents the same secure origin from being crawled twice under cosmetically different addresses.

Mental Model

Writing a phone number with the country code you didn't need to dial: 443 is the assumed extension, so adding it explicitly reaches the same line.

Project 260 Check Paginated Pages Self-Canonicalise

The Idea Behind It

Pages two and beyond of a listing series should self-canonicalise rather than fold into page one, because folding loses the deep links and content unique to later pages. This project maps each paginated URL to its declared canonical and flags any page whose canonical differs from itself. Page two pointing at page one is flagged BAD, while pages that name themselves pass the check.

What You'll Build & Learn

Confirm later pages of a paginated series each self-canonicalise.

- Deep pages should self-canonicalise
- Why folding into page one loses content
- Flagging mis-pointed pagination

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 259: fold a default-port duplicate url. Project 261 changes the surface problem while preserving the underlying method.

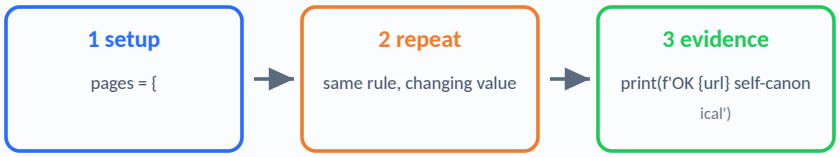
progress check

Reading target: check paginated pages self-canonicalise is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'OK {url} self-canonical')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if canon != url:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'OK {url} self-canonical')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if canon != url:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Paginated Pages Self-Canonicalise: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: pages, https, example, com, page. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = { | Core: if canon != url: | Evidence: print(f'OK {url} self-canonical')

🛠️ Reader to Writer Bridge

1. Read: explain check paginated pages self-canonicalise in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 259.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 261 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'OK {url} self-canonical')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check paginated pages self-canonicalise and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A paginated page whose canonical differs from its own URL is wrongly collapsing the series; each page in the sequence should name itself so its unique listings stay indexable.

```
pages = {
    'https://example.com/list?page=1': 'https://example.com/list?page=1',
    'https://example.com/list?page=2': 'https://example.com/list?page=1',
    'https://example.com/list?page=3': 'https://example.com/list?page=3',
}
for url in sorted(pages):
    canon = pages[url]
```

```
    if canon != url:
        print(f'BAD {url} -> {canon} (page folded into page 1; lost deep
links)')
    else:
        print(f'OK  {url} self-canonical')
```

```
output
OK  https://example.com/list?page=1 self-canonical
BAD https://example.com/list?page=2 -> https://example.com/list?page=1 (page folded
into page 1; lost deep links)
OK  https://example.com/list?page=3 self-canonical
```

How it works:

1. A dict maps three list-page URLs to their canonical targets
2. Page two's canonical points at page one while the others point at themselves
3. The loop compares each page's canonical to its own URL
4. A mismatch prints a BAD line warning of lost deep links
5. A match prints OK confirming self-canonicalisation

Try it:

1. Allow a view-all page as the legitimate canonical target
2. Verify each canonical target actually exists in the set
3. Report the count of mis-canonicalised deep pages
4. Fix page 2 to self-canonicalise and watch its verdict turn OK.

⚠ Common Mistakes

- Assuming pointing every page at page 1 is correct; that is the very anti-pattern this catches.
- Confusing rel-canonical with rel-next/prev, which are the right tools for a series.
- Not sorting the pages makes the report order depend on dict insertion order.

Skills gained dict iteration | self-canonical check | pagination audit

→ Where You'll Use This: Folding deep pagination into page one is a recurring mistake that buries product and article links; auditing that each series page self-canonicalises keeps that deeper content discoverable.

Mental Model

Like stapling chapters two through ten onto chapter one's cover: the library now thinks the book is one page and shelves the rest out of sight.

Project 261 Cluster All Near-Duplicates with Union-Find

The Idea Behind It

Near-duplicate relationships are transitive, so if A resembles B and B resembles C, all three belong in one group. This project scores every page pair by Jaccard over 2-word shingles and, when a pair clears the threshold, unions them with a union-find structure. After merging, each page is mapped to its cluster root, gathering three connected coat-sale pages into one cluster while the sourdough page stands alone.

What You'll Build & Learn

Merge transitively-linked near-duplicate pages into clusters with union-find.

- Near-duplicate links are transitive
- Union-find for clustering
- Merging connected pairs

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 260: check paginated pages self-canonicalise. Project 262 changes the surface problem while preserving the underlying method.

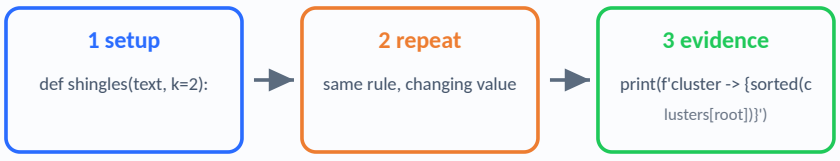
progress check

Reading target: cluster all near-duplicates with union-find is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def shingles(text, k=2):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print(f'cluster -> {sorted(clusters[root])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def shingles(text, k=2):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if len(a & b) / len(a | b) >= threshold:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'cluster -> {sorted(clusters[root])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(a & b) / len(a | b) >= threshold:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Cluster All Near-Duplicates with Union-Find: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: shingles, text, lower, split, tuple. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.

Layer	What to read
Evidence	Entry: def shingles(text, k=2): | Core: if len(a & b) / len(a | b) >= threshold: | Evidence: print(f'cluster -> {sorted(clusters[root])}')

🛠️ Reader to Writer Bridge

1. Read: explain cluster all near-duplicates with union-find in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 260.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 262 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'cluster -> {sorted(clusters[root])}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `def shingles(text, k=2):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of cluster all near-duplicates with union-find and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each near-duplicate pair unions its two roots, so transitively similar pages end up sharing one root; grouping by that root yields complete near-duplicate clusters, even when p1 and p3 are not directly similar.

```
def shingles(text, k=2):
    w = text.lower().split()
    return {tuple(w[i:i + k]) for i in range(len(w) - k + 1)}

pages = {
    'https://example.com/p1': 'big sale on winter coats now',
    'https://example.com/p2': 'big sale on winter coats today',
    'https://example.com/p3': 'big sale on winter jackets today',
    'https://example.com/p4': 'how to bake fresh sourdough bread',
}
threshold = 0.4
urls = sorted(pages)
parent = {u: u for u in urls}
```

```
    def find(x):
        while parent[x] != x:
            x = parent[x]
        return x
    for i in range(len(urls)):
        for j in range(i + 1, len(urls)):
            a, b = shingles(pages[urls[i]]), shingles(pages[urls[j]])
            if len(a & b) / len(a | b) >= threshold:
                parent[find(urls[j])] = find(urls[i])
    clusters = {}
    for u in urls:
        clusters.setdefault(find(u), []).append(u)
    for root in sorted(clusters):
        print(f'cluster -> {sorted(clusters[root])}')
```

```
output
cluster -> ['https://example.com/p1', 'https://example.com/p2',
'https://example.com/p3']
cluster -> ['https://example.com/p4']
```

How it works:

1. A shingles helper builds word-pair sets per page
2. parent maps each URL to itself as a starting forest
3. find walks parent pointers up to a cluster root
4. Each qualifying pair unions one root under the other
5. Grouping by find root yields the final clusters, one of three pages and one alone

Try it:

1. Add union-by-rank or path compression for speed on big sets
2. Lower the threshold and watch separate clusters merge
3. Print the chosen canonical for each resulting cluster
4. Raise the threshold to 0.6 and watch p3 split into its own cluster.

⚠ Common Mistakes

- Comparing only adjacent pages misses transitive links union-find is meant to catch.
- Pointing parents the wrong way (`parent[find(i)] = find(j)`) still works but must be consistent.
- Mutating `parent` while reading `find` mid-loop is fine here but path-compression must not corrupt roots.

Skills gained union-find | transitive merge | cluster build

→ Where You'll Use This: Union-find turns pairwise similarity flags into whole clusters of duplicates, the form SEO teams actually act on when they pick one canonical for a family of barely-different pages.

Mental Model

Like merging friend circles at a party: once you learn A knows B and B knows C, you draw one circle around all three rather than three loose pairs.

Project 262 Fingerprint a Page with SimHash-Lite

The Idea Behind It

SimHash maps a whole document to a single bitstring so that similar documents produce nearby fingerprints. This project hashes each token, then for every bit position adds one when the token's bit is set and subtracts one otherwise; the sign of each accumulated total fixes the output bit. The Hamming distance between two fingerprints estimates difference, and a one-bit gap here marks the pages as near-duplicates.

What You'll Build & Learn

Fingerprint a page with SimHash and measure difference by Hamming distance.

- SimHash yields one bitstring
- Voting bits with plus and minus one
- Hamming distance estimates difference

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 261: cluster all near-duplicates with union-find. Project 263 changes the surface problem while preserving the underlying method.

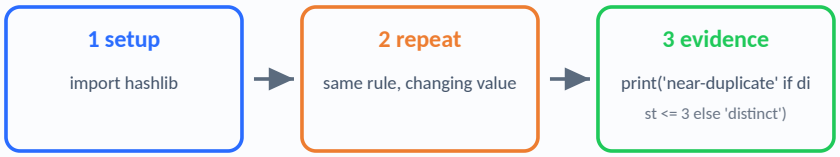
progress check

Reading target: fingerprint a page with simhash-lite is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import hashlib` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
- Evidence: `print('near-duplicate' if dist <= 3 else 'distinct')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import hashlib`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`v[i] += 1 if (h >> i) & 1 else -1`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('near-duplicate' if dist <= 3 else 'distinct')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `v[i] += 1 if (h >> i) & 1 else -1`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Fingerprint a Page with SimHash-Lite: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: hashlib, simhash, text, bits, tok. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import hashlib | Core: v[i] += 1 if (h >> i) & 1 else -1 | Evidence: print('near-duplicate' if dist <= 3 else 'distinct')

🛠️ Reader to Writer Bridge

1. Read: explain fingerprint a page with simhash-lite in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 261.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 263 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('near-duplicate' if dist <= 3 else 'distinct')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import hashlib` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of fingerprint a page with simhash-lite and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each token votes +1 or -1 per bit position from its SHA-1; the sign of each summed column sets that fingerprint bit, so swapping one word flips only a few bits and the XOR popcount measures the difference.

```
import hashlib

def simhash(text, bits=16):
    v = [0] * bits
    for tok in text.lower().split():
```

```
            h = int(hashlib.sha1(tok.encode()).hexdigest(), 16)
            for i in range(bits):
                v[i] += 1 if (h >> i) & 1 else -1
        out = 0
        for i in range(bits):
            if v[i] > 0:
                out |= (1 << i)
        return out

a = simhash('the quick brown fox jumps over')
b = simhash('the quick brown fox leaps over')
dist = bin(a ^ b).count('1')
print('simhash A:', format(a, '016b'))
print('simhash B:', format(b, '016b'))
print('hamming distance:', dist)
print('near-duplicate' if dist <= 3 else 'distinct')
```

```
output
simhash A: 1111101000000110
simhash B: 1111001000000110
hamming distance: 1
near-duplicate
```

How it works:

1. simhash initialises a vote vector of sixteen zeros
2. Each token is hashed, and for each bit the vote rises or falls by one
3. After voting, any position with a positive total sets that output bit
4. Two phrases differing in one word produce fingerprints one bit apart
5. The Hamming distance is the popcount of the XOR, and three or fewer means near-duplicate

Try it:

1. Widen to sixty-four bits for finer discrimination
2. Weight tokens by frequency before voting
3. Compare an unrelated document and confirm a large Hamming distance
4. Change two words instead of one and watch the Hamming distance grow.

⚠ Common Mistakes

- Counting set bits with `bin(...).count('1')` on the XOR is the Hamming distance; counting on each value separately is wrong.
- Too few bits (e.g. 4) makes unrelated documents collide on the same fingerprint.
- Forgetting tokens vote -1 for zero bits collapses the fingerprint toward all-ones.

Skills gained bit voting hamming distance document fingerprint

→ Where You'll Use This: SimHash is the technique large crawlers use to bucket near-duplicate web pages at scale, letting them detect templated or scraped content without comparing full text pairwise.

Mental Model

Like a fingerprint where each ridge votes; documents about the same thing leave nearly the same print, and you count the few ridges that differ.

Project 263 Run a Full Canonical and Duplicate Audit

The Idea Behind It

This capstone fuses the chapter's three checks into one pass over a page set. For each page it counts words, hashes the normalised body, and accumulates flags: non-self-canonical when the canonical differs from the URL, duplicate-of when the hash was already seen, and thin when the word count falls under a threshold. The first seen body becomes the keeper, so later identical pages are reported as duplicates of it.

What You'll Build & Learn

Run canonical, exact-duplicate and thin-content checks together in a single pass.

- Combining three audits in one loop
- First-seen hash becomes the keeper
- Accumulating flags per page

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 262: fingerprint a page with simhash-lite. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run a full canonical and duplicate audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import hashlib` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.

- Evidence: `print(f'{url}: {flags if flags else ["clean"]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import hashlib`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
2. Apply the rule	`if m['canonical'] != url:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{url}: {flags if flags else ["clean"]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Canonicalization, Duplicate and Thin Content becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m['canonical'] != url:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run a Full Canonical and Duplicate Audit: connect the surface task to the chapter skill, check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages.
Names	Watch these names first: hashlib, pages, https, example, com. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import hashlib | Core: if m['canonical'] != url: | Evidence: print(f'{url}: {flags if flags else ["clean"]}')

🛠️ Reader to Writer Bridge

1. Read: explain run a full canonical and duplicate audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 262.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{url}: {flags if flags else ["clean"]}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import hashlib` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run a full canonical and duplicate audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

One sorted pass collects every issue per page: a canonical that is not self-referencing, a content hash already seen (an exact duplicate of the first holder), and a body below the thin threshold.

```
import hashlib
pages = {
    'https://example.com/a': {'canonical': 'https://example.com/a', 'body': 'A
full article about indexing and canonical tags that runs well past the thin
threshold.'},
    'https://example.com/b': {'canonical': 'https://example.com/a', 'body': 'A
full article about indexing and canonical tags that runs well past the thin
threshold.'},
    'https://example.com/c': {'canonical': 'https://example.com/d', 'body':
'Too short.'},
}
thin = 8
seen = {}
for url in sorted(pages):
    m = pages[url]
    wc = len(m['body'].split())
    h = hashlib.sha1(' '.join(m['body'].lower().split()).encode()).hexdigest()
    flags = []
    if m['canonical'] != url:
        flags.append('non-self-canonical')
    if h in seen:
        flags.append(f'duplicate-of:{seen[h]}')
    else:
        seen[h] = url
    if wc < thin:
        flags.append(f'thin({wc}w)')
    print(f'{url}: {flags if flags else ["clean"]}')
```

```
output
https://example.com/a: ['clean']
https://example.com/b: ['non-self-canonical', 'duplicate-of:https://example.com/a']
https://example.com/c: ['non-self-canonical', 'thin(2w)']
```

How it works:

1. Each page maps to a dict with a canonical target and a body
2. The loop counts words and hashes the normalised body with SHA-1
3. A canonical not equal to the URL adds the non-self-canonical flag
4. A previously-seen hash adds a duplicate-of flag naming the keeper
5. A word count under the thin threshold adds a thin flag, and clean pages report clean

Try it:

1. Add a soft-404 marker check to the same pass
2. Lower the thin threshold and rescore the set
3. Output the flags as JSON rows for a downstream dashboard

4. Add a Jaccard near-duplicate check alongside the exact-hash check.

⚠ Common Mistakes

- Recording `seen[h]` before checking membership marks the first page as a duplicate of itself.
- Iterating the raw dict instead of `sorted(pages)` makes which URL holds a hash first nondeterministic.
- Hashing without normalising whitespace would split pages a and b into separate, non-duplicate hashes.

Skills gained `hashlib.sha1` `multi-check loop` `flag accumulation`

→ Where You'll Use This: A unified audit pass like this is the shape of a real crawl report: each row carries every problem found, so an SEO can triage canonical, duplicate and thin-content issues from one table.

⚡ Mental Model

Like an MOT inspection that checks brakes, lights and tyres in one drive-through, stamping each fault on a single sheet rather than three separate visits.

✅ Chapter Progress Checkpoint

What this chapter added: after 38 mini-projects, Canonicalization, Duplicate and Thin Content should feel like a reusable Python move: check canonical consistency, detect duplicates with shingling/jaccard, and flag thin pages. *chapter checkpoint*

1 Read
Open any project from Projects 226-263 and mark `setup`, `rule`, and `evidence` without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 8 (Site Speed and Core Web Vitals) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

7 questions · 4 categories · graded

2 Recall | 4 Meaning | 1 Structure | 0 Transfer

7.1.Q1 Recall MEDIUM

Turn back to Project 226 (Check a Canonical Is Self-Referencing). Without running it, what exactly does it print?

7.1.Q2 Meaning MEDIUM Turn back to Project 237 (Compute a Boilerplate-to-Content Ratio). Without running it, what exactly does it print?

7.1.Q3 Structure MEDIUM Turn back to Project 231 (Strip Tracking Params from a URL). Without running it, what exactly does it print?

7.1.Q4 Meaning EASY What does an ordinary, healthy page's canonical tag normally point at?

- A Its parent category page
- B Itself (a self-referencing canonical)
- C The homepage
- D A noindex copy

7.1.Q5 Recall EASY Why can a SHA-1 content hash detect exact duplicates but not near-duplicates?

- A SHA-1 is too slow
- B A single changed character produces a completely different digest, so similar-but-not-identical pages never collide
- C Hashes only work on URLs
- D Near-duplicates have the same byte length

7.1.Q6 Meaning EASY During URL normalization, why do you lowercase the host but not the path?

- A The path is shorter
- B Hostnames are case-insensitive but paths are usually case-sensitive on the server
- C Search engines ignore the host
- D It is faster

7.1.Q7 Meaning MEDIUM Two pages share 3 shingles, and the union of their shingle sets is 5. What is their Jaccard similarity, and what does it measure?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 7 – Canonicalization, Duplicate and Thin Content

7.1.Q1. `self-referencing canonical: OK`

7.1.Q2. `boilerplate 5/10 = 0.5 -> ok`

7.1.Q3. `clean: https://example.com/sale?id=42&page=2` then `dropped: 2`

7.1.Q4. B a self-referencing canonical (canonical == its own URL) is the baseline; it tells the engine "this is the representative, index me."

7.1.Q5. B hashing is all-or-nothing; one different word changes the whole digest, so you need a similarity measure (shingles + Jaccard) for near-duplicates.

7.1.Q6. B `Example.com` and `example.com` are the same server, but `/Page` and `/page` can be different resources, so only the host is safe to lowercase.

7.1.Q7. 3/5 = 0.6. Jaccard similarity is the size of the shingle intersection over the union — the fraction of content the two pages share, from 0 (nothing in common) to 1 (identical).

Chapter 8 · Site Speed and Core Web Vitals

Score LCP, INP and CLS against thresholds, set performance budgets, and weigh page resources.

Chapter Skill Map

What this chapter adds: In Site Speed and Core Web Vitals, the focus is score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources. You practise it across 38 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 8 assumes the reading habits from Chapter 7: Canonicalization, Duplicate and Thin Content. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 9, Structured Data and Rich Results, reuses this chapter's discipline with a new kind of problem.

1 **Projects 264-275**	2 **Projects 276-287**	3 **Projects 288-301**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Speed is three felt moments, each graded against a threshold: LCP loads, INP responds, CLS stays still.

how fast it shows, how fast it reacts, how much it jumps

"Fast" is not one number — it is three different experiences a visitor has in the first few seconds. Largest Contentful Paint (LCP) asks *how long until the main thing I came for is on screen?* Interaction to Next Paint (INP) asks *when I tap or click, how long until something visibly happens?* And Cumulative Layout Shift (CLS) asks *did the page hold still, or did a button jump out from under my finger?* Google calls these the *Core Web Vitals*, and they are a ranking signal because they predict whether a human stays or bounces.

Each vital is a raw measurement turned into a verdict by two thresholds. Below the first you are good; between the two you need improvement; above the second you are poor. The whole of this chapter is built on mapping a measured number to one of those three bands, so the very first tool is exactly that — an LCP grader:

```
def grade_lcp(seconds):
    if seconds <= 2.5:
        return 'good'
    if seconds <= 4.0:
        return 'needs-improvement'
    return 'poor'

for s in [2.1, 3.4, 4.8]:
    print(f'LCP {s}s -> {grade_lcp(s)}')
```

```
output
LCP 2.1s -> good
LCP 3.4s -> needs-improvement
LCP 4.8s -> poor
```

The same two-threshold shape repeats for every vital — only the numbers change. INP is graded in milliseconds (≤200 good, ≤500 needs-improvement), CLS is a unitless score (≤0.1 good, ≤0.25 needs-improvement). Memorise the three pairs and you can read any vitals report at a glance.

LCP 2.5/4.0s · INP 200/500ms · CLS 0.1/0.25

Two rules tie it together. A page only `passes Core Web Vitals` when *all three* are in the good band at once — one poor metric fails the whole assessment. And the number that counts is not your laptop's reading: Google judges the `p75` — the 75th percentile of *real* visits gathered in the field (the Chrome User Experience Report, or CrUX). The rest of the chapter is the engineering behind those numbers: page weight, render-blocking resources and the performance budgets that keep them in check.

LCP (LARGEST CONTENTFUL PAINT)

How long until the largest visible element — usually the hero image or headline — finishes rendering. Bands: good ≤ `2.5s`, needs-improvement ≤ `4.0s`, poor > `4.0s`.

INP (INTERACTION TO NEXT PAINT)

Responsiveness: the time from a user interaction (tap, click, key) to the next frame the browser paints. Replaced FID in March 2024. Bands: good ≤ `200ms`, needs-improvement ≤ `500ms`, poor > `500ms`.

CLS (CUMULATIVE LAYOUT SHIFT)

Visual stability: a unitless score summing how much content jumps around unexpectedly. Each shift contributes `impact-fraction × distance-fraction`. Bands: good ≤ `0.1`, needs-improvement ≤ `0.25`, poor > `0.25`.

P75 (75TH PERCENTILE)

The field value Google grades against: 75% of real visits are at least this fast. Using p75 instead of the average means a few quick loads cannot hide many slow ones.

FIELD VS LAB DATA

Field data (CrUX) is collected from real Chrome users on their real devices and networks. Lab data is a single synthetic test (e.g. Lighthouse) on a simulated device — repeatable but optimistic.

PERFORMANCE BUDGET

A hard cap on a measurable cost — total transfer size, per-type KB, request count or a Lighthouse score — that a page is not allowed to exceed. A budget turns "keep it fast" into a pass/fail gate.

RENDER-BLOCKING RESOURCE

A CSS file or synchronous `<script>` in the document head that the browser must download and process before it can paint, directly delaying first paint and LCP.

Three vitals, three graders — same shape, different numbers

Every vital is the same algorithm: compare the measurement to two thresholds and return a band. Memorising the thresholds is most of the battle. INP — the responsiveness metric that replaced FID in 2024 — is graded in milliseconds:

```
def grade_inp(ms):
    if ms <= 200:
        return 'good'
    if ms <= 500:
        return 'needs-improvement'
    return 'poor'

for ms in [150, 480, 640]:
    print(f'INP {ms}ms -> {grade_inp(ms)}')
```

```
output
INP 150ms -> good
INP 480ms -> needs-improvement
INP 640ms -> poor
```

CLS is the odd one out: it is unitless — not seconds, not milliseconds, but a score. A layout that never moves scores 0; a page where a banner shoves the article down scores well above 0.25. Because the numbers are tiny, the thresholds feel surprisingly tight: 0.1 is good, but 0.25 is already poor.

```
def grade_cls(score):
    if score <= 0.1:
        return 'good'
```

```
    if score <= 0.25:
        return 'needs-improvement'
    return 'poor'

for c in [0.05, 0.12, 0.33]:
    print(f'CLS {c} -> {grade_cls(c)}')
```

```
output
CLS 0.05 -> good
CLS 0.12 -> needs-improvement
CLS 0.33 -> poor
```

Passing means all three good at once

A page does not pass Core Web Vitals by averaging its scores or being good "on balance". It passes only when every vital sits in the good band simultaneously — a single poor metric fails the whole assessment. Rather than write three separate cascades, drive the check from a thresholds dict so the limits live in one place:

```
metrics    = {'lcp': 2.3, 'inp': 175, 'cls': 0.07}
thresholds = {'lcp': 2.5, 'inp': 200, 'cls': 0.1}
results = {k: metrics[k] <= thresholds[k] for k in thresholds}
for k in sorted(results):
    print(f'{k}: {"good" if results[k] else "needs work"}')
print('Core Web Vitals:', 'PASS' if all(results.values()) else 'FAIL')
```

```
output
cls: good
inp: good
lcp: good
Core Web Vitals: PASS
```

When a page *fails*, the same dict tells you where to spend effort: list the metrics over their threshold and fix the first one. There is no point shaving milliseconds off INP if it is LCP that is in the red. *diagnose before you optimise*

The number Google grades is the p75 of real users

Your one fast test on a fast laptop is lab data — useful, but optimistic. Google grades the field data: real Chrome users on real phones and networks, summarised as the 75th percentile. The p75 means three-quarters of visits were at least this fast, so a handful of quick loads cannot paper over a slow majority. Here is the interpolated p75 of a list of LCP samples:

```
samples = [1.8, 2.0, 2.1, 2.4, 2.6, 3.0, 3.1, 3.5, 4.0, 4.2]
ordered = sorted(samples)
rank = 0.75 * (len(ordered) - 1)
lo = int(rank)
frac = rank - lo
```

```
p75 = ordered[lo] + frac * (ordered[lo + 1] - ordered[lo]) if frac else ordered[lo]
print(f'p75 LCP = {round(p75, 3)}s')
```

```
output
p75 LCP = 3.4s
```

The average of those samples is about 2.9s — comfortably under 4.0s — yet the p75 is 3.4s, firmly in *needs-improvement*. That gap is the whole reason Google chose a percentile: the slow tail is the experience that loses you visitors. CrUX takes this one step further — a metric passes in the field when at least 75% of samples land in the good band.

Speed comes from weight — so cap it with a budget

Vitals are symptoms; page weight and render-blocking resources are the cause. The remedy is a performance budget: a hard KB cap a page may not exceed. Sum the bytes a visitor downloads and compare — if you are over, the headroom goes negative and the build fails the gate:

```
resources = {'html': 18, 'css': 42, 'js': 310, 'images': 540, 'fonts': 90}
budget_kb = 900
total = sum(resources.values())
verdict = 'within budget' if total <= budget_kb else 'OVER budget'
print(f'total = {total} KB, budget = {budget_kb} KB -> {verdict}')
print(f'headroom = {budget_kb - total} KB')
```

```
output
total = 1000 KB, budget = 900 KB -> OVER budget
headroom = -100 KB
```

A single page budget hides *which* resource type blew it, so split it per type — JS, CSS, images, fonts each get their own cap and the culprit is obvious. Separately, CSS and synchronous scripts in the head are render-blocking: the browser cannot paint until they download and run, which directly delays LCP. Counting them tells you exactly what is standing between the visitor and first paint:

```
resources = [
    {'url': 'app.js',       'blocking': True},
    {'url': 'analytics.js', 'blocking': False},
    {'url': 'main.css',     'blocking': True},
    {'url': 'font.css',     'blocking': True},
    {'url': 'lazy.js',      'blocking': False},
]
blocking = [r['url'] for r in resources if r['blocking']]
print('render-blocking:', len(blocking))
for u in sorted(blocking):
    print(' -', u)
```

```
output
```

```
render-blocking: 3
 - app.js
 - font.css
 - main.css
```

Defer the analytics, inline the critical CSS, lazy-load below-the-fold images, and the blocking count drops — which is what moves LCP from poor back into good. Budget and blocking-count are the two levers; the vitals are the readout.

cut weight and unblock paint, and the vitals follow

Example 1

A page reports field p75 values LCP 2.3s, INP 480ms, CLS 0.07. Does it pass Core Web Vitals, and if not, what do you fix first?

SOLUTION

1. Grade each against its bands. LCP 2.3s ≤ 2.5 → good.
2. INP 480ms is above 200 but ≤ 500 → needs-improvement, so it is *not* in the good band.
3. CLS 0.07 ≤ 0.1 → good. Two good, one not — and passing requires all three good.
4. The only failing metric is INP, so that is where the work goes: break up long main-thread tasks and defer non-critical JavaScript.

FAIL — INP is the single metric out of the good band, so fix responsiveness first.

one not-good metric fails the whole page

Example 2

Ten real-user LCP samples (seconds): 2.0, 2.1, 2.2, 2.3, 2.4, 2.6, 2.8, 3.0, 3.4, 5.0. The average is 2.78s. Will Google call this page good?

SOLUTION

1. Google grades the p75, not the average, so the 2.78s average is irrelevant.
2. Sort the samples (already sorted) and find the 75th-percentile position: rank = 0.75 × (10 − 1) = 6.75.
3. Interpolate between index 6 (2.8) and index 7 (3.0): 2.8 + 0.75 × (3.0 − 2.8) = 2.8 + 0.15 = 2.95s.
4. Compare 2.95s to the LCP bands: it is above 2.5 but ≤ 4.0 → needs-improvement.

No — the p75 is 2.95s (needs-improvement), even though the average of 2.78s sits in the good band. The slow tail (that 5.0s visit) drags the percentile up.

average flatters; p75 tells the truth

Example 3

A page ships js 520 KB, css 80 KB, images 600 KB, fonts 140 KB against per-type budgets of js 400, css 100, images 500, fonts 150 KB. Which types are over, and what is the total overage?

SOLUTION

1. Diff each type against its cap: js 520 − 400 = +120 over; css 80 − 100 = under (ok).
2. images 600 − 500 = +100 over; fonts 140 − 150 = under (ok).
3. Two types breach their budgets: JS (+120 KB) and images (+100 KB).
4. Total overage = 120 + 100 = 220 KB — and JS is the bigger offender, so split the bundle and defer non-critical scripts first.

JS (+120 KB) and images (+100 KB) are over, 220 KB total — a per-type budget pinpoints the category to attack, which a single page budget would have hidden. *per-type budgets name the culprit*

Mental-Math Shortcut

Loads, responds, stays still — at the 75th percentile. LCP is how fast the main content *loads* (≤2.5s good), INP is how fast the page *responds* to a tap (≤200ms good), CLS is whether it *stays still* (≤0.1 good). All three must be good to pass, and Google judges the p75 of real users — not your fast laptop, not the average. When a page is slow, look at weight and render-blocking resources; the vitals are the readout, not the root cause.

⚠ Common Mistakes

- Trusting one Lighthouse run (lab data on a fast simulated device) and ignoring the p75 field data Google actually grades — the field numbers are usually worse, especially on mobile.
- Treating Core Web Vitals as a score to average. A page passes only when LCP, INP and CLS are all in the good band; a single poor metric fails the whole assessment.
- Optimising the average load time instead of the slow tail. Google uses the 75th percentile, so the worst quarter of visits is exactly what you must fix.
- Forgetting CLS is unitless and the bands are tiny — a 0.3 score "sounds small" but is firmly poor; a single late-loading ad or un-sized image can blow it.
- Adding render-blocking CSS or synchronous scripts to the head without measuring — each one delays first paint and pushes LCP toward the poor band.

This chapter contains 38 hands-on mini-projects (Projects 264–301). Type each one out, run it, and experiment.

Project 264 Classify LCP Against Its Thresholds

The Idea Behind It

Largest Contentful Paint records the moment a page's biggest visible element finishes rendering. This project turns a raw seconds value into Google's verdict using two cutoffs: at or under 2.5 means good, at or under 4.0 means needs-improvement, and anything larger is poor. The grade_lcp function encodes those bands as ordered comparisons so every sampled path lands in exactly one bucket.

What You'll Build & Learn

Map an LCP timing to Google's good, needs-improvement, or poor band for several sampled URLs.

- The three official LCP thresholds
- Mapping a number to a labelled band
- Ordered if-checks form a cascade

Skill Roadmap

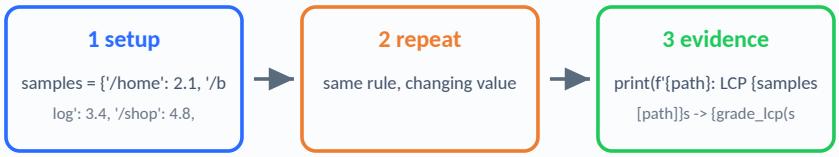
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 265 changes the surface problem while preserving the underlying method. *progress check*

Reading target: classify lcp against its thresholds is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `samples = {'/home': 2.1, '/blog': 3.4, '/shop': 4.8, '/about': 2.5}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{path}: LCP {samples[path]}s -> {grade_lcp(samples[path])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`samples = {'/home': 2.1, '/blog': 3.4, '/shop': 4.8, '/about': 2.5}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`if s <= 2.5:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{path}: LCP {samples[path]}s -> {grade_lcp(samples[path])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if s <= 2.5:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Classify LCP Against Its Thresholds: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: samples, home, blog, shop, about. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: samples = {'/home': 2.1, '/blog': 3.4, '/shop': 4.8, '/about': 2.5} | Core: if s <= 2.5: | Evidence: print(f'{path}: LCP {samples[path]}s -> {grade_lcp(samples[path])}')

🛠️ Reader to Writer Bridge

1. Read: explain classify lcp against its thresholds in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 265 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: LCP {samples[path]}s -> {grade_lcp(samples[path])}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `samples = {'/home': 2.1, '/blog': 3.4, '/shop': 4.8, '/about': 2.5}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify lcp against its thresholds and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A simple cascade of `if/elif/else` over the two thresholds buckets every LCP value into its Core Web Vitals band, exactly as PageSpeed Insights colour-codes the metric.

```
samples = {'/home': 2.1, '/blog': 3.4, '/shop': 4.8, '/about': 2.5}
def grade_lcp(s):
    if s <= 2.5:
        return 'good'
    if s <= 4.0:
```

```
        return 'needs-improvement'
    return 'poor'
for path in sorted(samples):
    print(f'{path}: LCP {samples[path]}s -> {grade_lcp(samples[path])}')
```

```
output
/about: LCP 2.5s -> good
/blog: LCP 3.4s -> needs-improvement
/home: LCP 2.1s -> good
/shop: LCP 4.8s -> poor
```

How it works:

1. A samples dict pairs four paths with their LCP seconds, from /home at 2.1 up to /shop at 4.8.
2. grade_lcp checks the boundaries top down: 2.5 returns good, then 4.0 returns needs-improvement, else poor.
3. Looping over sorted(samples) prints each path, its LCP, and the grade, so /shop at 4.8 reads as poor.

Try it:

1. Add a /pricing path at exactly 4.0 seconds and confirm it grades as needs-improvement, not poor.
2. Return a tuple of grade plus the gap to the good cutoff so callers see how far over budget a page is.
3. Replace the cascade with a list of (limit, label) pairs scanned in a loop.
4. Add a page at exactly 4.0s and confirm it grades needs-improvement, not poor.

⚠ Common Mistakes

- Using `<` instead of `<=` mis-grades a page that lands exactly on 2.5s as needs-improvement.
- Comparing the float `4.0` for equality somewhere instead of using ranges drops the boundary case.
- Forgetting to sort the dict keys before printing makes the per-page output order vary.

Skills gained LCP thresholds | cascading conditionals | dict iteration

→ Where You'll Use This: Every Lighthouse and PageSpeed report colour-codes LCP this way, so the banding logic underpins any dashboard or alert you build to flag slow-loading templates.

Mental Model

Like a swimmer's lane assignment at a meet: the clock reads a time, and that one number drops you into the fast, middle, or slow heat.

Project 265 Classify INP Against Its Thresholds

The Idea Behind It

Interaction to Next Paint took over from First Input Delay in 2024 and measures, in milliseconds, how sluggishly a page responds when someone taps or clicks. The grade_inp function applies INP's own cutoffs: 200ms or less is good, 500ms or less is needs-improvement, beyond that is poor. Each sampled interaction path gets classified independently against those latency boundaries.

What You'll Build & Learn

Convert per-page INP latencies in milliseconds into the good, needs-improvement, or poor responsiveness band.

- INP replaced FID as a responsiveness metric
- The 200 and 500 millisecond cutoffs
- Reusing the cascade pattern for a new unit

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 264: classify lcp against its thresholds. Project 266 changes the surface problem while preserving the underlying method.

progress check

Reading target: classify inp against its thresholds is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

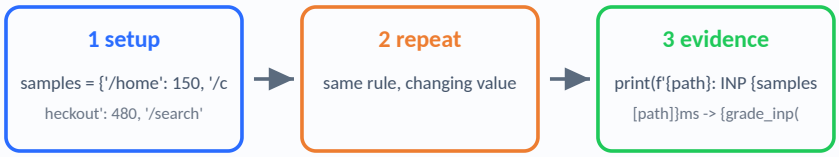
- Setup: `samples = {'/home': 150, '/checkout': 480, '/search': 640, '/cart': 200}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{path}: INP {samples[path]}ms -> {grade_inp(samples[path])}')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`samples = {'/home': 150, '/checkout': 480, '/search': 640, '/cart': 200}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`if ms <= 200:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{path}: INP {samples[path]}ms -> {grade_inp(samples[path])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if ms <= 200:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Classify INP Against Its Thresholds: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: samples, home, checkout, search, cart. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: samples = {'/home': 150, '/checkout': 480, '/search': 640, '/cart': 200} | Core: if ms <= 200: | Evidence: print(f'{path}: INP {samples[path]}ms -> {grade_inp(samples[path])}')

🛠️ Reader to Writer Bridge

1. Read: explain classify inp against its thresholds in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 264.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 266 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: INP {samples[path]}ms -> {grade_inp(samples[path])}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `samples = {'/home': 150, '/checkout': 480, '/search': 640, '/cart': 200}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify inp against its thresholds and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

INP uses the same three-band cascade as LCP but with millisecond thresholds; the heaviest-interaction pages (search, checkout) usually fail first.

```
samples = {'/home': 150, '/checkout': 480, '/search': 640, '/cart': 200}
def grade_inp(ms):
    if ms <= 200:
        return 'good'
    if ms <= 500:
```

```
        return 'needs-improvement'
    return 'poor'
for path in sorted(samples):
    print(f'{path}: INP {samples[path]}ms -> {grade_inp(samples[path])}')
```

```
output
/cart: INP 200ms -> good
/checkout: INP 480ms -> needs-improvement
/home: INP 150ms -> good
/search: INP 640ms -> poor
```

How it works:

1. The samples dict holds four paths with INP in milliseconds, from /home at 150 to /search at 640.
2. grade_inp tests 200 first for good, then 500 for needs-improvement, otherwise poor.
3. Iterating sorted(samples) prints each path with its millisecond reading and grade; /search at 640 prints poor.

Try it:

1. Add a /menu path at 500ms and verify it grades as needs-improvement rather than poor.
2. Track which paths are poor in a list and print only those needing urgent attention.
3. Generalise grade_inp to accept the cutoffs as arguments so the same code grades any latency metric.
4. Change /search to 500ms and confirm it grades needs-improvement, not poor.

⚠ Common Mistakes

- Mixing up INP's 200/500ms bands with LCP's 2.5/4.0s bands grades everything wrong.
- Storing INP as seconds instead of milliseconds silently passes every page.
- Using `<` on the 200 boundary mis-grades /cart at exactly 200ms.

Skills gained INP thresholds | latency grading | millisecond units

→ Where You'll Use This: INP is the responsiveness pillar of Core Web Vitals, so this grading is what you reach for when triaging slow checkout or search interactions that frustrate users.

Mental Model

Think of a waiter's response time: under 200ms feels instant, half a second feels acceptable, and beyond that the table starts tapping the menu.

Project 266 Classify CLS Against Its Thresholds

The Idea Behind It

Cumulative Layout Shift is a unitless score that captures how much a page jumps around as it loads. Because its bands are tiny, 0.1 or under for good and 0.25 or under for needs-improvement, even a fraction of a point matters. The grade_cls function reads each sampled score and routes it into the matching stability band, showing how a /gallery score of 0.33 lands in poor.

What You'll Build & Learn

Classify CLS visual-stability scores into the good, needs-improvement, or poor band for several pages.

- CLS is a unitless stability score
- The 0.1 and 0.25 boundaries are small
- Why tiny shifts still fail the band

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 265: classify inp against its thresholds. Project 267 changes the surface problem while preserving the underlying method.

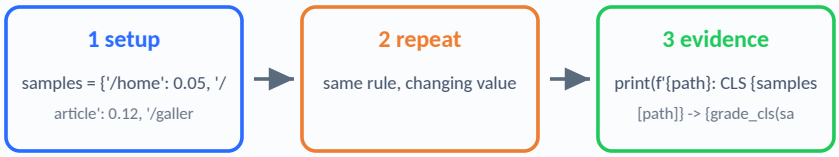
progress check

Reading target: classify cls against its thresholds is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `samples = {'/home': 0.05, '/article': 0.12, '/gallery': 0.33, '/landing': 0.25}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{path}: CLS {samples[path]} -> {grade_cls(samples[path])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`samples = {'/home': 0.05, '/article': 0.12, '/gallery': 0.33, '/landing': 0.25}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`if c <= 0.1:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{path}: CLS {samples[path]} -> {grade_cls(samples[path])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if c <= 0.1:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Classify CLS Against Its Thresholds: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: samples, home, article, gallery, landing. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: samples = {'/home': 0.05, '/article': 0.12, '/gallery': 0.33, '/landing... | Core: if c <= 0.1: | Evidence: print(f'{path}: CLS {samples[path]} -> {grade_cls(samples[path])}')

🛠️ Reader to Writer Bridge

1. Read: explain classify cls against its thresholds in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 265.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 267 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: CLS {samples[path]} -> {grade_cls(samples[path])}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `samples = {'/home': 0.05, '/article': 0.12, '/gallery': 0.33, '/landing': 0.25}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify cls against its thresholds and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

CLS is a ratio, not a time, so the thresholds are tiny decimals; the same cascade pattern applies, and a gallery of late-loading images is a classic poor performer.

```
samples = {'/home': 0.05, '/article': 0.12, '/gallery': 0.33, '/landing': 0.25}
def grade_cls(c):
    if c <= 0.1:
        return 'good'
    if c <= 0.25:
```

```
        return 'needs-improvement'
    return 'poor'
for path in sorted(samples):
    print(f'{path}: CLS {samples[path]} -> {grade_cls(samples[path])}')
```

```
output
/article: CLS 0.12 -> needs-improvement
/gallery: CLS 0.33 -> poor
/home: CLS 0.05 -> good
/landing: CLS 0.25 -> needs-improvement
```

How it works:

1. A samples dict maps four paths to CLS scores ranging from /home at 0.05 to /gallery at 0.33.
2. grade_cls compares against 0.1 then 0.25, falling through to poor for anything higher.
3. The sorted loop prints path, score, and grade, so /landing at 0.25 reads as needs-improvement at the boundary.

Try it:

1. Add a /press path at 0.1 and confirm the boundary value still grades as good.
2. Print a star next to any path scoring poor so the worst offenders stand out.
3. Round each score to two decimals before grading to mirror how reporting tools display CLS.
4. Set /landing to 0.26 and watch it tip from needs-improvement into poor.

⚠ Common Mistakes

- Reading 0.25 as a percentage and dividing by 100 mis-grades every page.
- Treating the 0.1 boundary with `<` mis-grades /home if it sat exactly on 0.1.
- Printing the raw float chain without sorting keys gives nondeterministic order.

Skills gained CLS thresholds | fractional comparisons | stability scoring

→ Where You'll Use This: Layout-shift complaints, ad slots or images loading without reserved space, are diagnosed by exactly this banding when you audit a template's visual stability.

Mental Model

Like rating how much a table wobbles: a hair of movement is fine, a noticeable rock is annoying, and a real lurch spills your coffee.

Project 267 Decide the Overall Core Web Vitals Pass

The Idea Behind It

Core Web Vitals is an all-or-nothing assessment: a page passes only when LCP, INP, and CLS are each inside their good band simultaneously. This project hardcodes one metric snapshot and joins three comparisons with logical and, so a single boolean decides the verdict. Because every term must be true, one weak metric drags the whole page to FAIL.

What You'll Build & Learn

Combine three metric checks with logical and to produce a single overall PASS or FAIL verdict.

- All three vitals must pass together
- Logical and chains the conditions
- One failure fails the whole assessment

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 266: classify cls against its thresholds. Project 268 changes the surface problem while preserving the underlying method.

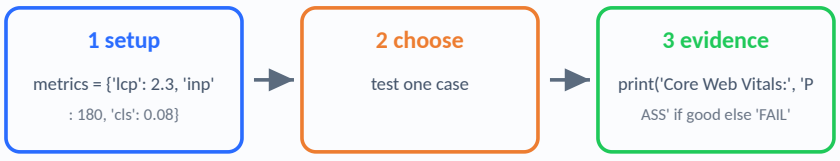
progress check

Reading target: decide the overall core web vitals pass is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `metrics = {'lcp': 2.3, 'inp': 180, 'cls': 0.08}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('Core Web Vitals:', 'PASS' if good else 'FAIL')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`metrics = {'lcp': 2.3, 'inp': 180, 'cls': 0.08}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`print('Core Web Vitals:', 'PASS' if good else 'FAIL')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('Core Web Vitals:', 'PASS' if good else 'FAIL')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('Core Web Vitals:', 'PASS' if good else 'FAIL')`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Decide the Overall Core Web Vitals Pass: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: metrics, lcp, inp, cls, good. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: metrics = {'lcp': 2.3, 'inp': 180, 'cls': 0.08} | Core: print('Core Web Vitals:', 'PASS' if good else 'FAIL') | Evidence: print('Core Web Vitals:', 'PASS' if good else 'FAIL')

🛠️ Reader to Writer Bridge

1. Read: explain decide the overall core web vitals pass in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 266.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 268 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('Core Web Vitals:', 'PASS' if good else 'FAIL')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `metrics = {'lcp': 2.3, 'inp': 180, 'cls': 0.08}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of decide the overall core web vitals pass and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Combining the three good-band checks with `and` enforces the all-three rule Google applies; a single laggard metric flips the verdict to FAIL.

```
metrics = {'lcp': 2.3, 'inp': 180, 'cls': 0.08}
good = metrics['lcp'] <= 2.5 and metrics['inp'] <= 200 and metrics['cls'] <=
0.1
print('Core Web Vitals:', 'PASS' if good else 'FAIL')
```

```
output
Core Web Vitals: PASS
```

How it works:

1. A metrics dict carries lcp 2.3, inp 180, and cls 0.08, all comfortably inside their good bands.
2. The good variable is the and of three threshold comparisons against 2.5, 200, and 0.1.

3. A conditional expression prints PASS because every comparison is true; flipping any value to a worse number would print FAIL.

Try it:

1. Bump inp to 250 and watch the combined verdict flip to FAIL even though LCP and CLS still pass.
2. Print which specific metric broke the pass rather than just PASS or FAIL.
3. Wrap the check in a function that takes a metrics dict and returns the boolean.
4. Bump `lcp` to 2.6 and watch the whole verdict flip to FAIL.

⚠ Common Mistakes

- Using `or` instead of `and` passes a page that is good on only one metric.
- Comparing INP in seconds against the 200ms threshold gives a false pass.
- Hard-coding the thresholds in three places makes them drift out of sync over time.

Skills gained boolean and | pass/fail logic | combined thresholds

→ Where You'll Use This: This single-page gate is the smallest unit of a Core Web Vitals report and the building block for site-wide pass-rate dashboards used in SEO health checks.

⚡ Mental Model

Like a three-key safe: every key must turn at once. One stuck key and the door stays shut no matter how good the other two are.

Project 268 Score All Three Vitals From a Threshold Table

💡 The Idea Behind It

Instead of three hardcoded comparisons, this version stores the limits in a thresholds dict and builds a results dict with a comprehension that compares each metric to its matching cap. all() then collapses the per-metric booleans into one verdict. Keeping the limits in a single table means adding a new metric is a one-line edit rather than another if-branch.

What You'll Build & Learn

Drive the per-metric pass check from a thresholds table and reduce the results with all().

- Data-driven thresholds beat hardcoded branches
- Dict comprehension pairs metric to verdict
- all() reduces booleans to one result

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 267: decide the overall core web vitals pass. Project 269 changes the surface problem while preserving the underlying method.

progress check

Reading target: score all three vitals from a threshold table is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `metrics = {'lcp': 2.3, 'inp': 175, 'cls': 0.07}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('Core Web Vitals:', 'PASS' if passed else 'FAIL')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output.

visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`metrics = {'lcp': 2.3, 'inp': 175, 'cls': 0.07}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`print(f'{k}: {"good" if results[k] else "needs work"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('Core Web Vitals:', 'PASS' if passed else 'FAIL')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{k}: {"good" if results[k] else "needs work"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score All Three Vitals From a Threshold Table: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: metrics, lcp, inp, cls, thresholds. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: metrics = {'lcp': 2.3, 'inp': 175, 'cls': 0.07} | Core: print(f'{k}: {"good" if results[k] else "needs work"}') | Evidence: print('Core Web Vitals:', 'PASS' if passed else 'FAIL')

Reader to Writer Bridge

1. Read: explain score all three vitals from a threshold table in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 267.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 269 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('Core Web Vitals:', 'PASS' if passed else 'FAIL')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `metrics = {'lcp': 2.3, 'inp': 175, 'cls': 0.07}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score all three vitals from a threshold table and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A dict comprehension compares each metric to its threshold in lockstep, and `all()` over the booleans yields the overall pass, so adding a metric is a one-line change.

```
metrics = {'lcp': 2.3, 'inp': 175, 'cls': 0.07}
thresholds = {'lcp': 2.5, 'inp': 200, 'cls': 0.1}
results = {k: metrics[k] <= thresholds[k] for k in thresholds}
passed = all(results.values())
for k in sorted(results):
    print(f'{k}: {"good" if results[k] else "needs work"}')
print('Core Web Vitals:', 'PASS' if passed else 'FAIL')
```

```
output
cls: good
inp: good
lcp: good
Core Web Vitals: PASS
```

How it works:

1. metrics and a parallel thresholds dict share the keys lcp, inp, and cls.
2. A dict comprehension builds results, mapping each key to whether its metric is at or under its threshold.
3. all(results.values()) decides PASS, and a sorted loop prints good or needs work per metric before the verdict line.

Try it:

1. Add a ttfb key to both dicts and confirm the comprehension and all() pick it up automatically.
2. Print the count of passing metrics alongside the overall verdict.
3. Switch all() to any() and describe how the meaning of the verdict changes.
4. Add an 'fcp' metric and threshold and watch the table grow automatically.

⚠ Common Mistakes

- Iterating over `metrics` instead of `thresholds` breaks if the metrics dict has extra keys.
- Assuming every threshold is an upper bound; a 'higher is better' metric would need inverting.
- Printing the booleans dict unsorted yields nondeterministic key order.

Skills gained `dict comprehension` `all() reduction` `table-driven logic`

→ Where You'll Use This: Configuration-as-data is how real monitoring tools store metric limits, so this pattern scales when teams add new web-performance signals without rewriting the checker.

Mental Model

Like a checklist clipped to a clipboard: every item has its own tick box, and the form is approved only when every box is ticked.

Project 269 Check Total Transfer Against a Budget

The Idea Behind It

A performance budget caps the total kilobytes a page is allowed to ship. This project sums the per-type transfer sizes from a resources dict, compares the total against a fixed budget, and reports whether the page is within or over. It also prints headroom as budget minus total, which goes negative when the page busts the cap, here landing at -100 KB.

What You'll Build & Learn

Sum per-type transfer sizes, compare to a kilobyte budget, and report verdict plus headroom.

- A performance budget caps page weight
- sum() totals dict values quickly
- Negative headroom signals a busted budget

Skill Roadmap

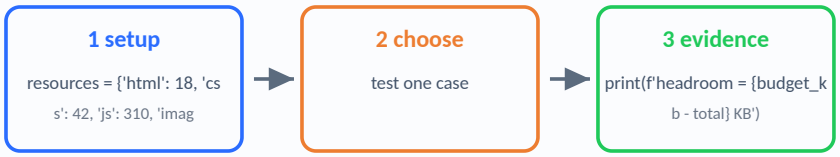
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 268: score all three vitals from a threshold table. Project 270 changes the surface problem while preserving the underlying method. *progress check*

Reading target: check total transfer against a budget is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `resources = {'html': 18, 'css': 42, 'js': 310, 'images': 540, 'fonts': 90}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'headroom = {budget_kb - total} KB')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`resources = {'html': 18, 'css': 42, 'js': 310, 'images': 540, 'fonts': 90}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`verdict = 'within budget' if total <= budget_kb else 'OVER budget'`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(f'headroom = {budget_kb - total} KB')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'within budget' if total <= budget_kb else 'OVER budget'`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Check Total Transfer Against a Budget: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: resources, html, css, js, images. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: resources = {'html': 18, 'css': 42, 'js': 310, 'images': 540, 'fonts':... | Core: verdict = 'within budget' if total <= budget_kb else 'OVER budget' | Evidence: print(f'headroom = {budget_kb - total} KB')

🛠️ Reader to Writer Bridge

1. Read: explain check total transfer against a budget in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 268.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 270 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'headroom = {budget_kb - total} KB')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `resources = {'html': 18, 'css': 42, 'js': 310, 'images': 540, 'fonts': 90}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check total transfer against a budget and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Summing the resource dict's values gives the page weight, and subtracting from the budget surfaces the headroom (or, when negative, the overage).

```
resources = {'html': 18, 'css': 42, 'js': 310, 'images': 540, 'fonts': 90}
budget_kb = 900
total = sum(resources.values())
verdict = 'within budget' if total <= budget_kb else 'OVER budget'
print(f'total transfer = {total} KB, budget = {budget_kb} KB -> {verdict}')
print(f'headroom = {budget_kb - total} KB')
```

```
output
total transfer = 1000 KB, budget = 900 KB -> OVER budget
headroom = -100 KB
```

How it works:

1. The resources dict lists html, css, js, images, and fonts sizes that sum to 1000 KB.
2. sum(resources.values()) gives the total, compared against budget_kb of 900.
3. A conditional sets the verdict to OVER budget, and headroom prints as budget minus total, showing -100 KB.

Try it:

1. Lower the budget to 800 KB and confirm the headroom and verdict both reflect the tighter cap.
2. Print the percentage of the budget consumed in addition to the verdict.
3. Raise an exception instead of printing when the total exceeds the budget, mimicking a CI failure.
4. Raise the budget to 1100 KB and watch the verdict flip to within budget.

⚠ Common Mistakes

- Summing the dict keys instead of `.values()` adds strings and raises a TypeError.
- Mixing KB and bytes between the budget and the sizes makes the comparison meaningless.
- Reporting only pass/fail and hiding the headroom number that tells you how close you are.

Skills gained performance budgets summing values budget gating

→ Where You'll Use This: Performance budgets are enforced in CI to block bloated builds, so this gate is the seed of a check that fails a deploy when a page outgrows its weight allowance.

Mental Model

Like a suitcase with an airline weight limit: you weigh everything packed, and if the scale tips past the allowance you are repacking or paying extra.

Project 270 Sum Resource Weights From a Manifest

The Idea Behind It

Before any budget can be checked you need the raw page weight, and that means totalling every asset a visitor downloads. This project models the manifest as a list of (url, type, size) tuples and uses a generator inside sum() to add up only the size field via tuple unpacking. The result is both the resource count and the combined transfer in kilobytes.

What You'll Build & Learn

Total the size field across a list of resource tuples to find the full page transfer weight.

- A manifest as a list of tuples
- Tuple unpacking ignores fields with underscore
- Generator expression feeds sum()

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 269: check total transfer against a budget. Project 271 changes the surface problem while preserving the underlying method.

progress check

Reading target: sum resource weights from a manifest is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `resources = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.

- Evidence: `print(f'{len(resources)} resources, {total} KB total')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`resources = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`total = sum(size for _, _, size in resources)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{len(resources)} resources, {total} KB total')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `total = sum(size for _, _, size in resources)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Sum Resource Weights From a Manifest: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: resources, app, js, vendor, style. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: resources = [| Core: total = sum(size for _, _, size in resources) | Evidence: print(f'{len(resources)} resources, {total} KB total')

🛠️ Reader to Writer Bridge

1. Read: explain sum resource weights from a manifest in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 269.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 271 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{len(resources)} resources, {total} KB total')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `resources = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sum resource weights from a manifest and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Unpacking each tuple and summing the size field gives total weight; counting the list length gives the request count, the two headline numbers of any waterfall.

```
resources = [
    ('app.js', 'js', 280),
    ('vendor.js', 'js', 150),
```

```
    ('style.css', 'css', 60),
    ('hero.jpg', 'image', 420),
    ('logo.png', 'image', 30),
]
total = sum(size for _, _, size in resources)
print(f'{len(resources)} resources, {total} KB total')
```

```
output
5 resources, 940 KB total
```

How it works:

1. resources is a list of five tuples, each holding a filename, a type, and a size in kilobytes.
2. sum(size for _, _, size in resources) unpacks each tuple, discarding url and type, and totals only the sizes.
3. The print reports len(resources) as the count and the total, giving 5 resources and 940 KB.

Try it:

1. Sum only the JavaScript entries by adding a type filter to the generator expression.
2. Find the single heaviest resource using max with a key on the size field.
3. Convert the total to megabytes and print it rounded to two decimal places.
4. Print the average resource size (`total / len(resources)`).

⚠ Common Mistakes

- Unpacking the tuple in the wrong order sums the type strings instead of sizes.
- Counting `len(resources)` as bytes instead of as the number of requests.
- Forgetting that a real manifest may list the same URL twice and double-count it.

Skills gained tuple unpacking | generator in sum | page weight

→ Where You'll Use This: Network panels and HAR exports hand you exactly this kind of resource list, so summing sizes is the first step in any byte-level performance audit.

Mental Model

Like adding up grocery receipt line items: you only care about the price column, scanning past the names and aisles to reach the total at the bottom.

Project 271 Count Render-Blocking Resources

The Idea Behind It

CSS and synchronous JavaScript placed in the document head stall first paint until they download and run. This project models each resource as a dict with a blocking flag, filters the list to the blocking URLs with a

comprehension, and reports how many delay rendering. Printing the sorted offenders names exactly which files sit on the critical path.

What You'll Build & Learn

Filter a resource list to the render-blocking entries and count and name what delays first paint.

- Render-blocking assets stall first paint
- Filter a list by a boolean flag
- List comprehension extracts matching URLs

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 270: sum resource weights from a manifest. Project 272 changes the surface problem while preserving the underlying method.

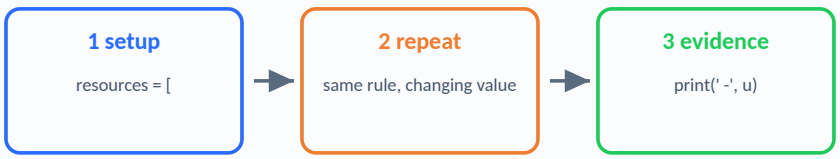
progress check

Reading target: count render-blocking resources is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `resources = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(' -', u)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`resources = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`blocking = [r['url'] for r in resources if r['blocking']]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' –', u)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `blocking = [r['url'] for r in resources if r['blocking']]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Render-Blocking Resources: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: resources, url, app, js, blocking. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: resources = [| Core: blocking = [r['url'] for r in resources if r['blocking']] | Evidence: print(' -', u)

🛠️ Reader to Writer Bridge

1. Read: explain count render-blocking resources in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 270.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 272 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' -', u)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `resources = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count render-blocking resources and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Filtering the resource list on the `blocking` flag isolates the critical resources, and sorting the URLs makes the report deterministic.

```
resources = [
    {'url': 'app.js', 'blocking': True},
    {'url': 'analytics.js', 'blocking': False},
    {'url': 'main.css', 'blocking': True},
    {'url': 'font.css', 'blocking': True},
    {'url': 'lazy.js', 'blocking': False},
]
blocking = [r['url'] for r in resources if r['blocking']]
print('render-blocking:', len(blocking))
for u in sorted(blocking):
    print(' -', u)
```

```
output
render-blocking: 3
 - app.js
 - font.css
 - main.css
```

How it works:

1. resources is a list of dicts, each with a url and a blocking boolean.

2. A comprehension keeps the url where blocking is True, collecting app.js, main.css, and font.css.
3. The code prints the count of three, then loops over the sorted blocking list to name each render-blocking file.

Try it:

1. Add an async script with blocking False and confirm it is excluded from the count.
2. Group the blocking resources by file extension to see whether CSS or JS dominates.
3. Invert the filter to list the non-blocking resources that are safe to leave alone.
4. Mark `font.css` as non-blocking and watch the count drop to 2.

⚠ Common Mistakes

- Treating `async`/`defer` scripts as blocking inflates the count.
- Counting non-blocking analytics as blocking because it is also JavaScript.
- Printing the filtered set unsorted so the URL order changes run to run.

Skills gained render blocking | boolean filtering | list comprehension

→ Where You'll Use This: Eliminating render-blocking resources, by deferring scripts or inlining critical CSS, is a top Lighthouse recommendation, and this count is how you measure the problem.

Mental Model

Like a single-file doorway where a few people insist on entering first: until they squeeze through, everyone behind them waits and the room stays empty.

Project 272 Total Image Weight and Find the Heaviest

The Idea Behind It

Images are typically the heaviest slice of a page and the most common LCP element, so they deserve first attention. This project sums the values of an images dict to get total image weight and uses max with a key over the sorted keys to name the single biggest file. Identifying hero.jpg at 480 KB points straight at the asset worth optimising.

What You'll Build & Learn

Total all image sizes and name the single heaviest image to prioritise optimisation effort.

- Images often dominate page weight
- max with a key finds the heaviest
- Sorting keys makes ties deterministic

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 271: count render-blocking resources. Project 273 changes the surface problem while preserving the underlying method.

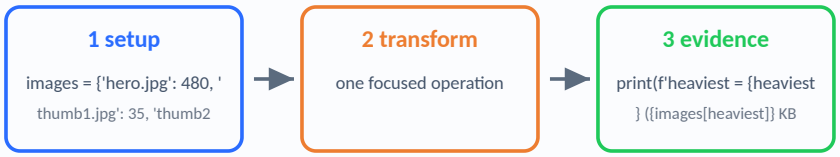
progress check

Reading target: total image weight and find the heaviest is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `images = {'hero.jpg': 480, 'thumb1.jpg': 35, 'thumb2.jpg': 40, 'banner.png': 220}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is deliberately small: Python stores a value, applies one clear operation, and prints evidence. That tiny chain is the reading pattern larger programs still use.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'heaviest = {heaviest} ({images[heaviest]} KB)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`images = {'hero.jpg': 480, 'thumb1.jpg': 35, 'thumb2.jpg': 40, 'banner.png': 220}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`total = sum(images.values())`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(f'heaviest = {heaviest} ({images[heaviest]} KB)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `total = sum(images.values())`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Total Image Weight and Find the Heaviest: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: images, hero, jpg, thumb1, thumb2. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: images = {'hero.jpg': 480, 'thumb1.jpg': 35, 'thumb2.jpg': 40, 'banner.... | Core: total = sum(images.values()) | Evidence: print(f'heaviest = {heaviest} ({images[heaviest]} KB)')

Reader to Writer Bridge

1. Read: explain total image weight and find the heaviest in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 271.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 273 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'heaviest = {heaviest} ({images[heaviest]} KB)')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `images = {'hero.jpg': 480, 'thumb1.jpg': 35, 'thumb2.jpg': 40, 'banner.png': 220}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of total image weight and find the heaviest and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Summing the image dict's values gives total image payload, and `max` over the sorted keys with a size key picks the heaviest file deterministically.

```
images = {'hero.jpg': 480, 'thumb1.jpg': 35, 'thumb2.jpg': 40, 'banner.png':
220}
total = sum(images.values())
heaviest = max(sorted(images), key=lambda k: images[k])
print(f'total image weight = {total} KB')
print(f'heaviest = {heaviest} ({images[heaviest]} KB)')
```

```
output
total image weight = 775 KB
heaviest = hero.jpg (480 KB)
```

How it works:

1. An images dict pairs four filenames with their kilobyte sizes.
2. sum(images.values()) totals the weight to 775 KB.
3. max(sorted(images), key=lambda k: images[k]) returns hero.jpg, and the print reports it at 480 KB as the heaviest.

Try it:

1. Print the top two heaviest images instead of just one by sorting and slicing.
2. Report what percentage of total image weight the heaviest file represents.
3. Add a tiny icon at 2 KB and confirm it never wins the max comparison.
4. Add a 600 KB photo and watch the heaviest file change.

⚠ Common Mistakes

- Calling `max(images)` returns the alphabetically last key, not the heaviest file.

- Passing `images.values()` to `max` loses which filename owns the size.
- Not sorting before `max` lets a tie resolve unpredictably.

Skills gained `max with key` `image weight` `lambda keys`

→ **Where You'll Use This:** Image-heavy LCP elements are the usual culprit behind slow loads, so spotlighting the biggest file is the practical start of a compress-and-resize plan.

Mental Model

Like packing for a trip and spotting the one bulky coat that eats half the bag: shrink that and everything else fits more easily.

Project 273 Split Page Bytes Between JS and CSS

The Idea Behind It

Knowing the JavaScript-to-CSS ratio tells you whether a page is script-heavy, which often explains slow interactivity. This project sums the resource sizes separately by type using two filtered generators, then turns each subtotal into a percentage of the combined total with rounded integer arithmetic. The 83 percent JS share frames the page as decidedly script-dominant.

What You'll Build & Learn

Sum bytes by type and express the JS and CSS shares as rounded percentages of the total.

- JS-to-CSS ratio signals script weight
- Filtered generators sum by type
- Percentage as rounded integer division

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 272: total image weight and find the heaviest. Project 274 changes the surface problem while preserving the underlying method. *progress check*

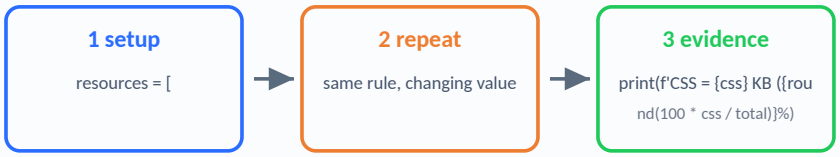
Reading target: split page bytes between js and css is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `resources = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'CSS = {css} KB ({round(100 * css / total)}%)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`resources = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`js = sum(s for _, t, s in resources if t == 'js')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'CSS = {css} KB ({round(100 * css / total)}%)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `js = sum(s for _, t, s in resources if t == 'js')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Split Page Bytes Between JS and CSS: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: resources, app, js, vendor, style. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: resources = [| Core: js = sum(s for _, t, s in resources if t == 'js') | Evidence: print(f'CSS = {css} KB ({round(100 * css / total)}%)')

🛠️ Reader to Writer Bridge

1. Read: explain split page bytes between js and css in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 272.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 274 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'CSS = {css} KB ({round(100 * css / total)}%)')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `resources = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split page bytes between js and css and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Two filtered sums split the payload by type, and rounding `100 * part / total` gives whole-number percentages that always add to roughly 100.

```
resources = [
    ('app.js', 'js', 300),
    ('vendor.js', 'js', 180),
    ('style.css', 'css', 90),
    ('reset.css', 'css', 10),
]
js = sum(s for _, t, s in resources if t == 'js')
css = sum(s for _, t, s in resources if t == 'css')
total = js + css
print(f'JS = {js} KB ({round(100 * js / total)}%)')
print(f'CSS = {css} KB ({round(100 * css / total)}%)')
```

```
output
JS = 480 KB (83%)
CSS = 100 KB (17%)
```

How it works:

1. resources is a list of tuples tagged js or css.
2. Two generators sum sizes where the type matches js then css, giving 480 and 100 KB.
3. Each is divided by the combined total, multiplied by 100, and rounded, printing 83% JS and 17% CSS.

Try it:

1. Add an image category and compute its share alongside JS and CSS.
2. Guard against a zero total so the percentage math does not divide by zero on an empty manifest.
3. Print a simple text bar whose length reflects each type's percentage.
4. Add a third type (fonts) and divide the budget three ways.

⚠ Common Mistakes

- Dividing by `js` or `css` alone instead of their combined total skews the percentages.
- Integer-dividing before multiplying by 100 collapses the ratio to 0.
- Rounding both shares independently can make them sum to 99 or 101.

Skills gained `type filtering` `percentage math` `round()`

→ **Where You'll Use This:** Script-heavy pages tend to suffer the worst INP and TBT, so this ratio guides whether your performance work should target JavaScript or styling first.

⚡ Mental Model

Like splitting a restaurant bill by category: once you see drinks were eighty percent of it, you know where the night's cost really went.

Project 274 Grade Time To First Byte

💡 The Idea Behind It

Time To First Byte measures how long the server takes to send the first byte of a response, and it feeds directly into LCP. This project applies a common rule of thumb, fast at 200ms or under, moderate at 600ms or under, slow beyond that, via an if/elif/else cascade. A reading of 640ms therefore lands in slow, pointing at server or network latency.

What You'll Build & Learn

Grade a server's TTFB in milliseconds as fast, moderate, or slow using a three-way cascade.

- TTFB is server response latency
- TTFB feeds into the LCP timing
- if/elif/else builds a three-way grade

🧭 Skill Roadmap

`Skill step:` adds one small variation while the chapter pattern is still new. It builds on Project 273: split page bytes between js and css. Project 275 changes the surface problem while preserving the underlying method.

progress check

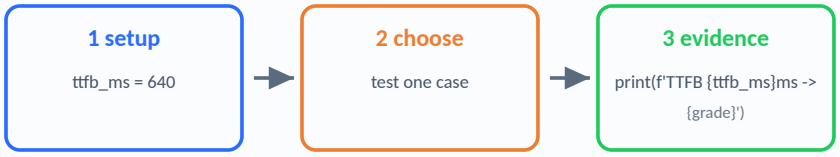
`Reading target:` grade time to first byte is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `ttfb_ms = 640` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- `Writing move:` this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.

- Evidence: `print(f'TTFB {ttfb_ms}ms -> {grade}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`ttfb_ms = 640`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`if ttfb_ms <= 200:`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(f'TTFB {ttfb_ms}ms -> {grade}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if ttfb_ms <= 200:`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Grade Time To First Byte: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: ttfb_ms, grade, fast, moderate, slow. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: ttfb_ms = 640 | Core: if ttfb_ms <= 200: | Evidence: print(f'TTFB {ttfb_ms}ms -> {grade}')

🛠️ Reader to Writer Bridge

1. Read: explain grade time to first byte in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 273.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 275 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'TTFB {ttfb_ms}ms -> {grade}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `ttfb_ms = 640` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of grade time to first byte and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

TTFB uses the same banded cascade as the vitals; a slow TTFB caps how good LCP can ever be because rendering cannot start until bytes arrive.

```
ttfb_ms = 640
if ttfb_ms <= 200:
    grade = 'fast'
elif ttfb_ms <= 600:
    grade = 'moderate'
else:
    grade = 'slow'
print(f'TTFB {ttfb_ms}ms -> {grade}')
```

```
output
TTFB 640ms -> slow
```

How it works:

1. ttfb_ms is set to 640 milliseconds.
2. The cascade checks 200 for fast, then 600 for moderate, otherwise slow.
3. Because 640 exceeds 600, grade becomes slow and the formatted print reports it.

Try it:

1. Wrap the logic in a grade_ttfb function so multiple readings can be graded in a loop.
2. Add an excellent band under 100ms to reward very fast servers.
3. Feed it a list of TTFB samples and print the worst reading and its grade.
4. Lower the value to 180 and confirm it grades fast.

⚠ Common Mistakes

- Treating TTFB as a Core Web Vital; it is a diagnostic input to LCP, not a graded vital.
- Using `<` on the 600ms boundary mis-grades a page sitting exactly at the limit.
- Confusing TTFB (server) with FCP (first paint) and grading the wrong number.

Skills gained TTFB grading | if/elif/else | server latency

→ Where You'll Use This: A slow TTFB usually points at backend or hosting issues, so this grading is the first triage when LCP is poor and you need to know whether the server is to blame.

Mental Model

Like timing how long a kitchen takes to acknowledge your order: quick is reassuring, a pause is tolerable, and a long silence means the meal is already running late.

Project 275 Detect Missing Cache Headers

The Idea Behind It

Static assets should carry caching headers so repeat visits skip the download entirely. This project inspects a response header dict for any of Cache-Control, Expires, or ETag, building present and missing lists with two comprehensions. With none of the three found, the asset is flagged NOT cacheable, exposing a missed caching opportunity.

What You'll Build & Learn

Check a response's headers for caching directives and report which are present, missing, and cacheable.

- Static assets need caching headers
- Membership test against a header dict
- Two comprehensions split present and missing

Skill Roadmap

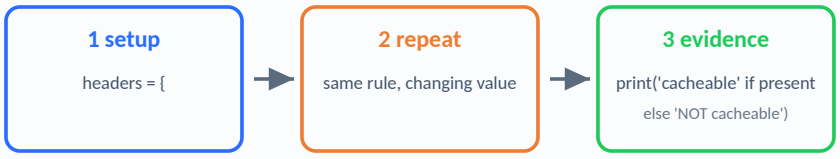
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 274: grade time to first byte. Project 276 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect missing cache headers is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `headers = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('cacheable' if present else 'NOT cacheable')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`headers = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`present = [h for h in cache_keys if h in headers]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('cacheable' if present else 'NOT cacheable')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `present = [h for h in cache_keys if h in headers]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Missing Cache Headers: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: headers, content, type, text, css. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: headers = { | Core: present = [h for h in cache_keys if h in headers] | Evidence: print('cacheable' if present else 'NOT cacheable')

🛠️ Reader to Writer Bridge

1. Read: explain detect missing cache headers in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 274.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 276 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('cacheable' if present else 'NOT cacheable')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `headers = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect missing cache headers and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Membership tests over a small set of caching headers partition them into present and missing; an asset with none of them forces a fresh download every visit.

```
headers = {
    'content-type': 'text/css',
    'content-length': '4200',
    'server': 'nginx',
}
cache_keys = ('cache-control', 'expires', 'etag')
present = [h for h in cache_keys if h in headers]
missing = [h for h in cache_keys if h not in headers]
print('caching headers present:', present)
print('missing:', missing)
print('cacheable' if present else 'NOT cacheable')
```

```
output
caching headers present: []
missing: ['cache-control', 'expires', 'etag']
NOT cacheable
```

How it works:

1. A headers dict holds content-type, content-length, and server but no caching keys.

2. cache_keys lists the three directives, and comprehensions partition them into present and missing by dict membership.
3. present is empty, so the final print reports NOT cacheable after listing the missing headers.

Try it:

1. Add a Cache-Control header and confirm the asset flips to cacheable.
2. Treat a Cache-Control of no-store as effectively not cacheable even though the header exists.
3. Run the check across a list of responses and tally how many lack caching headers.
4. Add `'cache-control': 'max-age=31536000'` and watch the verdict flip.

⚠ Common Mistakes

- Matching header names case-sensitively misses 'Cache-Control' if the dict uses 'cache-control'.
- Treating the presence of any header (like content-type) as proof of caching.
- Reporting 'cacheable' when only ETag is present but Cache-Control says no-store.

Skills gained cache headers | membership testing | list partition

→ Where You'll Use This: Missing Cache-Control is a frequent audit finding that slows repeat visits, so this check feeds directly into caching-policy recommendations for static files.

Mental Model

Like checking whether a library book has a due-date stamp: without one, every visitor has to re-borrow it from scratch each time.

Project 276 Detect Responses Missing Compression

The Idea Behind It

Text assets should travel gzip- or Brotli-compressed; an empty Content-Encoding means raw bytes are crossing the wire. This project walks a list of response dicts and uses a comprehension with a truthiness test on the encoding field to collect the uncompressed URLs. The two flagged responses are quick wins where enabling compression cuts transfer size.

What You'll Build & Learn

Scan responses for an empty Content-Encoding and list the uncompressed URLs as compression wins.

- Text assets should be compressed
- None encoding means raw bytes
- Truthiness filters out empty values

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 275: detect missing cache headers. Project 277 changes the surface problem while preserving the underlying method.

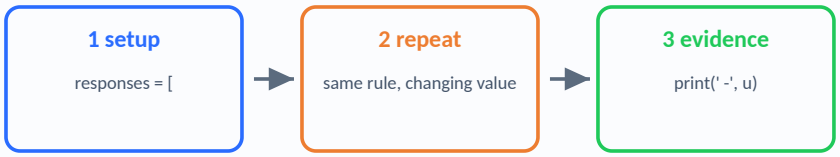
progress check

Reading target: detect responses missing compression is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `responses = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(' -', u)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`responses = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`uncompressed = [r['url'] for r in responses if not r['encoding']]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' -', u)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `uncompressed = [r['url'] for r in responses if not r['encoding']]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Responses Missing Compression: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: responses, url, app, js, encoding. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: responses = [| Core: uncompressed = [r['url'] for r in responses if not r['encoding']] | Evidence: print('-', u)

Reader to Writer Bridge

1. Read: explain detect responses missing compression in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 275.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 277 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' -', u)`. Then compare punctuation, spacing, and order.
2. Explain: point to `responses = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect responses missing compression and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Filtering responses whose `encoding` is falsy isolates the uncompressed assets, and sorting the URLs keeps the remediation list deterministic.

```
responses = [
    {'url': 'app.js', 'encoding': 'br'},
    {'url': 'style.css', 'encoding': 'gzip'},
    {'url': 'data.json', 'encoding': None},
    {'url': 'main.js', 'encoding': None},
]
uncompressed = [r['url'] for r in responses if not r['encoding']]
print('uncompressed responses:', len(uncompressed))
for u in sorted(uncompressed):
    print(' -', u)
```

```
output
uncompressed responses: 2
 - data.json
 - main.js
```

How it works:

1. responses is a list of dicts, two with br or gzip encoding and two with None.
2. The comprehension keeps the url where not r['encoding'] is true, catching the None entries.
3. It prints the count of two, then names data.json and main.js in sorted order.

Try it:

1. Treat an encoding of identity as uncompressed too, since it means no real compression.
2. Only flag uncompressed responses that are also text-based by adding a type field.
3. Count compressed versus uncompressed and print the ratio.
4. Set `data.json` to `'gzip'` and watch the uncompressed count fall to 1.

⚠ Common Mistakes

- Treating an empty string encoding as compressed because it is not None.
- Flagging already-compressed images (jpg/png) that gain nothing from gzip.
- Printing the set unsorted so the URL order changes between runs.

Skills gained compression detection | truthiness filter | Content-Encoding

→ Where You'll Use This: Enabling gzip or Brotli is among the easiest performance wins, so detecting uncompressed text responses turns directly into a server-config to-do list.

Mental Model

Like shipping clothes loose versus vacuum-sealed: the same garments take far less space compressed, and the uncompressed boxes are the obvious ones to redo.

Project 277 Count Un-Lazyloaded Images

The Idea Behind It

Below-the-fold images should carry loading="lazy" so they do not fight for bandwidth during the initial load. This project iterates a list of image dicts and uses a comprehension with an inequality test to collect any image whose loading attribute is not exactly lazy, including those set to eager or None. The three flagged images are eager downloads worth deferring.

What You'll Build & Learn

Find images lacking loading="lazy" and count and list the eager downloads worth deferring.

- Lazy loading defers offscreen images
- Inequality filter catches non-lazy values
- None and eager both count as not lazy

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 276: detect responses missing compression. Project 278 changes the surface problem while preserving the underlying method.

progress check

Reading target: count un-lazyloaded images is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `images = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(sorted(not_lazy))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`images = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`not_lazy = [i['src'] for i in images if i['loading'] != 'lazy']`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(sorted(not_lazy))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `not_lazy = [i['src'] for i in images if i['loading'] != 'lazy']`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Un-Lazyloaded Images: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: images, src, hero, jpg, loading. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: images = [| Core: not_lazy = [i['src'] for i in images if i['loading'] != 'lazy'] | Evidence: print(sorted(not_lazy))

🛠️ Reader to Writer Bridge

1. Read: explain count un-lazyloaded images in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 276.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 278 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(sorted(not_lazy))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `images = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of count un-lazyloaded images and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering images whose `loading` value is not 'lazy' catches both eager and missing attributes; the hero image is intentionally eager, the rest are candidates to lazy-load.

```
images = [
    {'src': 'hero.jpg', 'loading': 'eager'},
    {'src': 'photo1.jpg', 'loading': 'lazy'},
    {'src': 'photo2.jpg', 'loading': None},
    {'src': 'photo3.jpg', 'loading': None},
]
not_lazy = [i['src'] for i in images if i['loading'] != 'lazy']
print('images not lazy-loaded:', len(not_lazy))
print(sorted(not_lazy))
```

```
output
images not lazy-loaded: 3
['hero.jpg', 'photo2.jpg', 'photo3.jpg']
```

How it works:

1. images is a list of dicts with a loading value of eager, lazy, or None.
2. The comprehension keeps src where loading is not equal to lazy, catching hero.jpg, photo2.jpg, and photo3.jpg.
3. It prints the count of three and the sorted list of non-lazy sources.

Try it:

1. Keep the hero image eager on purpose by excluding it from the not-lazy list.
2. Also flag images using a deprecated lazyload class instead of the native attribute.
3. Report what fraction of all images are still loading eagerly.
4. Set photo2 and photo3 to 'lazy' and watch the count drop to 1.

⚠ Common Mistakes

- Treating a missing `loading` key as already lazy and skipping a real offender.
- Lazy-loading the LCP hero image, which actually delays the largest paint.
- Comparing against 'eager' only and missing images with no attribute at all.

Skills gained lazy loading inequality filter attribute audit

→ Where You'll Use This: Deferring offscreen images frees bandwidth for the LCP element, so this audit feeds the common recommendation to add lazy loading to long image-rich pages.

Mental Model

Like a buffet that only refills trays as guests reach them: filling every tray up front wastes food and crowds the line before anyone is there to eat.

Project 278 Inventory Preload and Preconnect Hints

The Idea Behind It

Resource hints such as preconnect, preload, and dns-prefetch let the browser warm up connections and fetch critical files early. This project uses a regular expression to pull the rel value out of each link tag in a markup string, then a Counter tallies how many of each hint type appear. The result is a quick inventory showing which hints a page actually declares.

What You'll Build & Learn

Parse link tags with a regex and tally each resource-hint type to inventory a page's connection warm-ups.

- Resource hints speed critical fetches
- Regex extracts the rel attribute value
- Counter tallies occurrences by kind

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 277: count un-lazyloaded images. Project 279 changes the surface problem while preserving the underlying method.

progress check

Reading target: inventory preload and preconnect hints is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `html = '''<link rel="preconnect" href="https://fonts.gstatic.com">` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.

- Evidence: `print(f'{hint}: {counts[hint]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`html = '''<link rel="preconnect" href="https://fonts.gstatic.com">`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`for hint in sorted(counts):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{hint}: {counts[hint]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for hint in sorted(counts):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Inventory Preload and Preconnect Hints: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: html, link, rel, preconnect, href. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: html = '''<link rel="preconnect" href="https://fonts.gstatic.com"> | Core: for hint in sorted(counts): | Evidence: print(f'{hint}: {counts[hint]}')

🛠️ Reader to Writer Bridge

1. Read: explain inventory preload and preconnect hints in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 277.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 279 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{hint}: {counts[hint]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `html = '''<link rel="preconnect" href="https://fonts.gstatic.com">` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of inventory preload and preconnect hints and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A regex over the `rel` attribute extracts every hint type, and a Counter tallies them so you can spot a page with zero hints or too many preloads.

```
html = '''<link rel="preconnect" href="https://fonts.gstatic.com">
<link rel="preload" href="/hero.jpg" as="image">
<link rel="stylesheet" href="/style.css">
<link rel="dns-prefetch" href="https://cdn.example.com">'''
import re
hints = re.findall(r'rel="(preconnect|preload|dns-prefetch|prefetch)"', html)
from collections import Counter
counts = Counter(hints)
for hint in sorted(counts):
    print(f'{hint}: {counts[hint]}')
```

```
output
dns-prefetch: 1
preconnect: 1
preload: 1
```

How it works:

1. An html string holds four link tags with rel values of preconnect, preload, stylesheet, and dns-prefetch.
2. re.findall with a capture group restricted to the hint names returns only the hint rels, skipping stylesheet.
3. A Counter over those matches is iterated in sorted order, printing dns-prefetch, preconnect, and preload each at one.

Try it:

1. Extend the regex to also capture the href so you can report which origin each hint targets.
2. Warn when a preload tag lacks a matching as attribute, which the browser ignores.
3. Count stylesheet links separately to see how many blocking CSS files load.
4. Add a second preload link and watch its count rise to 2.

⚠ Common Mistakes

- A greedy or loose regex would also match `rel="stylesheet"` and miscount.
- Matching `rel='preload'` with single quotes when the markup uses double quotes.
- Printing the Counter unsorted gives nondeterministic hint order.

Skills gained regex findall | Counter | resource hints

→ Where You'll Use This: Auditing resource hints reveals whether a page warms up connections to font and CDN origins, a common optimisation for cutting connection-setup latency.

Mental Model

Like a chef prepping ingredients before guests arrive: preconnect and preload are the mise en place that means the kitchen is ready the moment an order lands.

Project 279 Compare Field Data to Lab Data

The Idea Behind It

Lab tests run on a fast simulated device while field data from CrUX reflects real users on slow networks and phones, so the two rarely agree. This project pairs a lab and a field dict on the same metric keys, computes the field-minus-lab delta per metric, and prints a signed difference. The positive deltas reveal how optimistic the lab numbers were.

What You'll Build & Learn

Compute the per-metric gap between lab and field readings and print each as a signed delta.

- Lab data is optimistic synthetic testing
- Field data reflects real user conditions
- Signed deltas expose the gap

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 278: inventory preload and preconnect hints. Project 280 changes the surface problem while preserving the underlying method.

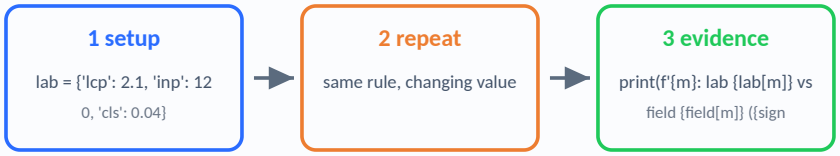
progress check

Reading target: compare field data to lab data is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `lab = {'lcp': 2.1, 'inp': 120, 'cls': 0.04}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{m}: lab {lab[m]} vs field {field[m]} ({sign}{delta})')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`lab = {'lcp': 2.1, 'inp': 120, 'cls': 0.04}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`sign = '+' if delta >= 0 else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{m}: lab {lab[m]} vs field {field[m]} ({sign}{delta})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sign = '+' if delta >= 0 else ''`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compare Field Data to Lab Data: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: lab, lcp, inp, cls, field. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: lab = {'lcp': 2.1, 'inp': 120, 'cls': 0.04} | Core: sign = '+' if delta >= 0 else '' | Evidence: print(f'{m}: lab {lab[m]} vs field {field[m]} ({sign}{delta})')

🛠️ Reader to Writer Bridge

1. Read: explain compare field data to lab data in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 278.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 280 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{m}: lab {lab[m]} vs field {field[m]} ({sign}{delta})')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `lab = {'lcp': 2.1, 'inp': 120, 'cls': 0.04}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compare field data to lab data and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Subtracting lab from field per metric and signing the delta exposes how much worse real users have it; a large positive gap means lab tests are flattering.

```
lab = {'lcp': 2.1, 'inp': 120, 'cls': 0.04}
field = {'lcp': 3.3, 'inp': 240, 'cls': 0.12}
for m in sorted(lab):
    delta = round(field[m] - lab[m], 3)
```

```
    sign = '+' if delta >= 0 else ''
    print(f'{m}: lab {lab[m]} vs field {field[m]} ({sign}{delta})')
```

```
output
cls: lab 0.04 vs field 0.12 (+0.08)
inp: lab 120 vs field 240 (+120)
lcp: lab 2.1 vs field 3.3 (+1.2)
```

How it works:

1. Two dicts, lab and field, share the keys lcp, inp, and cls.
2. For each sorted metric the code rounds field minus lab to three places and prepends a plus sign when non-negative.
3. It prints lab versus field with the signed delta, so lcp shows +1.2 and inp shows +120.

Try it:

1. Flag any metric where the field reading crosses into a worse band than the lab reading.
2. Express each delta as a percentage of the lab value instead of an absolute difference.
3. Add a verdict line summarising whether field data agrees with the lab overall.
4. Make field LCP better than lab and watch the sign go negative.

⚠ Common Mistakes

- Subtracting in the wrong order makes a real-world slowdown look like an improvement.
- Comparing metrics in different units (seconds vs milliseconds) across the two dicts.
- Printing raw float deltas without rounding produces noisy, unstable output.

Skills gained lab vs field | signed deltas | CrUX data

→ Where You'll Use This: Reconciling lab and field numbers is essential because Google ranks on field data, so this comparison stops you from celebrating a green lab score that real users never see.

Mental Model

Like comparing a car's advertised mileage to what you actually get in city traffic: the brochure number is best-case, and the real-world gap is what you plan around.

Project 280 Weight a Lighthouse-Style Performance Score

The Idea Behind It

Lighthouse blends several sub-metrics into one 0-100 performance score using fixed weights, with TBT and LCP carrying the most influence. This project multiplies each subscore by its weight, sums the products, and rounds

to a headline figure, then grades it green, orange, or red. Reproducing the weighted sum shows why a single slow metric like LCP at 60 can pull the score down to 67.

What You'll Build & Learn

Reproduce Lighthouse's weighted performance score from sub-metric scores and grade it by colour.

- Lighthouse weights sub-metrics differently
- Weighted sum produces one headline score
- TBT and LCP carry the most weight

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 279: compare field data to lab data. Project 281 changes the surface problem while preserving the underlying method.

progress check

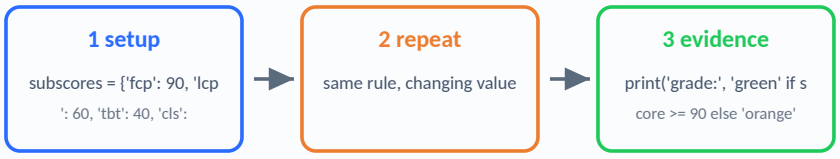
Reading target: weight a lighthouse-style performance score is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `subscores = {'fcp': 90, 'lcp': 60, 'tbt': 40, 'cls': 95, 'si': 70}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 else 'red')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`subscores = {'fcp': 90, 'lcp': 60, 'tbt': 40, 'cls': 95, 'si': 70}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 else 'red')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 else 'red')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 else 'red')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Weight a Lighthouse-Style Performance Score: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: subscores, fcp, lcp, tbt, cls. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: subscores = {'fcp': 90, 'lcp': 60, 'tbt': 40, 'cls': 95, 'si': 70} | Core: print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 els... | Evidence: print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 els...

🛠️ Reader to Writer Bridge

1. Read: explain weight a lighthouse-style performance score in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 279.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 281 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 else 'red')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `subscores = {'fcp': 90, 'lcp': 60, 'tbt': 40, 'cls': 95, 'si': 70}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of weight a lighthouse-style performance score and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Multiplying each sub-score by its weight and summing reproduces Lighthouse's weighted-average; the result maps to the familiar green/orange/red bands.

```
subscores = {'fcp': 90, 'lcp': 60, 'tbt': 40, 'cls': 95, 'si': 70}
weights = {'fcp': 0.10, 'lcp': 0.25, 'tbt': 0.30, 'cls': 0.25, 'si': 0.10}
score = sum(subscores[k] * weights[k] for k in weights)
print(f'performance score = {round(score)}')
print('grade:', 'green' if score >= 90 else 'orange' if score >= 50 else 'red')
```

```
output
performance score = 67
grade: orange
```

How it works:

1. A subscores dict and a weights dict share keys fcp, lcp, tbt, cls, and si.
2. A generator multiplies each subscore by its weight and sum() combines them, rounding to 67.
3. A chained conditional grades the result orange because it falls between 50 and 90.

Try it:

1. Set tbt to 95 and watch the heavily weighted improvement push the score over 90 to green.
2. Print each sub-metric's weighted contribution so the biggest drag is obvious.
3. Assert that the weights sum to 1.0 before computing, to catch a misconfigured table.
4. Drop the TBT sub-score to 10 and watch the headline score collapse.

⚠ Common Mistakes

- Using weights that do not sum to 1.0 silently rescales the whole score.
- Averaging the sub-scores unweighted ignores that TBT and LCP matter most.
- Rounding each weighted term before summing accumulates rounding error.

Skills gained weighted average | Lighthouse scoring | generator sum

→ Where You'll Use This: Understanding the weighting explains why teams chase TBT and LCP first, since those levers move the headline Lighthouse score the most in audits and CI gates.

Mental Model

Like a course grade where the final exam counts far more than a quiz: ace the small stuff but bomb the heavy-weight piece and your overall mark still sinks.

Project 281 Enforce a Per-Resource-Type Budget

The Idea Behind It

A single page-wide budget hides which resource type actually blew it, so this project sets a separate kilobyte cap per type. It walks a budgets dict alongside parallel usage figures, computes overage per category, and labels each as ok or OVER by an amount. The per-type breakdown pinpoints that images and JS, not CSS or fonts, busted their limits.

What You'll Build & Learn

Compare per-type usage against per-type budgets and report the overage for each resource category.

- Per-type budgets localise the problem
- Overage is usage minus the cap

- Parallel dicts share their keys

Skill Roadmap

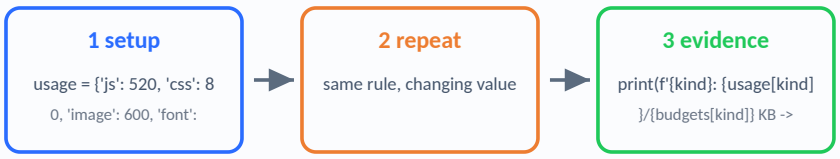
Skill step: moves you from recognition toward independent tracing. It builds on Project 280: weight a lighthouse-style performance score. Project 282 changes the surface problem while preserving the underlying method. *progress check*

Reading target: enforce a per-resource-type budget is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `usage = {'js': 520, 'css': 80, 'image': 600, 'font': 140}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{kind}: {usage[kind]}/{budgets[kind]} KB -> {status}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`usage = {'js': 520, 'css': 80, 'image': 600, 'font': 140}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`status = f'OVER by {over}' if over > 0 else 'ok'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'{kind}: {usage[kind]}/{budgets[kind]} KB -> {status}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `status = f'OVER by {over}' if over > 0 else 'ok'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Enforce a Per-Resource-Type Budget: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: usage, js, css, image, font. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: usage = {'js': 520, 'css': 80, 'image': 600, 'font': 140} | Core: status = f'OVER by {over}' if over > 0 else 'ok' | Evidence: print(f'{kind}: {usage[kind]}/{budgets[kind]} KB -> {status}')

Reader to Writer Bridge

1. Read: explain enforce a per-resource-type budget in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 280.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 282 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{kind}: {usage[kind]}/{budgets[kind]} KB -> {status}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `usage = {'js': 520, 'css': 80, 'image': 600, 'font': 140}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of enforce a per-resource-type budget and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Comparing each type's usage to its own cap reports the overage per category, so an over-budget JS bundle cannot hide behind under-budget CSS.

```
usage = {'js': 520, 'css': 80, 'image': 600, 'font': 140}
budgets = {'js': 400, 'css': 100, 'image': 500, 'font': 150}
for kind in sorted(budgets):
    over = usage[kind] - budgets[kind]
    status = f'OVER by {over}' if over > 0 else 'ok'
    print(f'{kind}: {usage[kind]}/{budgets[kind]} KB -> {status}')
```

```
output
css: 80/100 KB -> ok
font: 140/150 KB -> ok
image: 600/500 KB -> OVER by 100
js: 520/400 KB -> OVER by 120
```

How it works:

1. usage and budgets dicts share keys js, css, image, and font.
2. For each sorted kind the code computes usage minus budget and labels it OVER by that amount when positive, else ok.
3. The output shows css and font ok but image OVER by 100 and js OVER by 120.

Try it:

1. Sum all the overages into a single total-over-budget figure.

2. Add a percentage-over column so a small absolute miss on a tight budget still stands out.
3. Only print the categories that are over, suppressing the ok lines for a focused report.
4. Tighten the image budget to 400 and watch a second type go over.

⚠ Common Mistakes

- Comparing total usage to the sum of budgets masks a single type's overage.
- Missing a key in either dict raises a KeyError mid-check.
- Reporting 'ok' for a type that exactly equals its budget if you use `>=` wrongly.

Skills gained per-type budgets | overage math | parallel dicts

→ Where You'll Use This: Granular budgets are how performance teams hold each asset category accountable, so this report is what a CI budget check prints when it needs to name the offending type.

Mental Model

Like a household budget split into rent, food, and fun: a single total only says you overspent, but the categories reveal it was the dining-out line all along.

Project 282 Flag Oversized Images Above a Limit

The Idea Behind It

Any single image past a kilobyte limit is a candidate for compression or resizing. This project filters an images dict with a comprehension that keeps entries above a threshold, then prints the offenders sorted descending by size using a negated key. The two oversized files become a prioritised fix list, heaviest first.

What You'll Build & Learn

Filter images above a kilobyte limit and list the offenders sorted heaviest-first as a fix list.

- A size limit flags fixable images
- Dict comprehension filters by value
- Negated key sorts descending

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 281: enforce a per-resource-type budget. Project 283 changes the surface problem while preserving the underlying method.

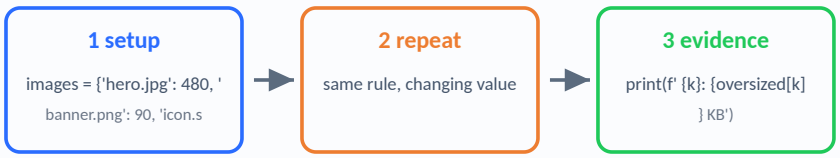
progress check

Reading target: flag oversized images above a limit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `images = {'hero.jpg': 480, 'banner.png': 90, 'icon.svg': 4, 'photo.jpg': 310}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f' {k}: {oversized[k]} KB')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`images = {'hero.jpg': 480, 'banner.png': 90, 'icon.svg': 4, 'photo.jpg': 310}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`oversized = {k: v for k, v in images.items() if v > limit_kb}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f' {k}: {oversized[k]} KB')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `oversized = {k: v for k, v in images.items() if v > limit_kb}`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Oversized Images Above a Limit: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: images, hero, jpg, banner, png. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: images = {'hero.jpg': 480, 'banner.png': 90, 'icon.svg': 4, 'photo.jpg'... | Core: oversized = {k: v for k, v in images.items() if v > limit_kb} | Evidence: print(f' {k}: {oversized[k]} KB')

🛠️ Reader to Writer Bridge

1. Read: explain flag oversized images above a limit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 281.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 283 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f' {k}: {oversized[k]} KB')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `images = {'hero.jpg': 480, 'banner.png': 90, 'icon.svg': 4, 'photo.jpg': 310}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag oversized images above a limit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A dict comprehension keeps only images above the limit, and sorting the offenders by descending size puts the worst hog first.

```
images = {'hero.jpg': 480, 'banner.png': 90, 'icon.svg': 4, 'photo.jpg': 310}
limit_kb = 200
oversized = {k: v for k, v in images.items() if v > limit_kb}
print(f'oversized (> {limit_kb} KB): {len(oversized)}')
for k in sorted(oversized, key=lambda x: -oversized[x]):
    print(f'  {k}: {oversized[k]} KB')
```

```
output
oversized (> 200 KB): 2
  hero.jpg: 480 KB
  photo.jpg: 310 KB
```

How it works:

1. An images dict pairs four files with sizes, and limit_kb is 200.
2. A dict comprehension keeps only the entries whose size exceeds the limit, catching hero.jpg and photo.jpg.
3. Sorting the keys by negative size prints hero.jpg at 480 then photo.jpg at 310, heaviest first.

Try it:

1. Make the threshold depend on image type, allowing larger hero images than thumbnails.
2. Print the total kilobytes that all oversized images could potentially save.
3. Lower the limit to 50 KB and watch how many more images get flagged.
4. Lower the limit to 50 KB and watch banner.png join the list.

⚠ Common Mistakes

- Using `>=` instead of `>` flags an image sitting exactly on the limit.
- Sorting ascending buries the worst offender at the bottom of the list.
- Hard-coding 200 in two places so the limit and the message can drift apart.

Skills gained dict comprehension threshold filter descending sort

→ Where You'll Use This: Oversized-image warnings are standard in performance audits, and ordering them by weight tells you which file to compress first for the biggest single byte saving.

Mental Model

Like a bouncer turning away oversized bags at a venue: only the ones past the size line get pulled aside, biggest first, for repacking.

Project 283 Estimate Savings From Compression

The Idea Behind It

Each text asset has a typical compression ratio, and multiplying its size by that ratio estimates the kilobytes gzip or Brotli would shave off. This project loops a list of resource dicts, computes the saving per file, accumulates a running total, and reports both. The 161 KB total quantifies the payoff of enabling compression before you touch a server config.

What You'll Build & Learn

Estimate per-resource and total compression savings by multiplying each size by its compression ratio.

- Compression ratio estimates byte savings
- Size times ratio is the per-file saving
- An accumulator builds the running total

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 282: flag oversized images above a limit. Project 284 changes the surface problem while preserving the underlying method.

progress check

Reading target: estimate savings from compression is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `resources = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.

- Evidence: `print(f'total estimated saving = {round(total_saving, 1)} KB')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`resources = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`for r in resources:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'total estimated saving = {round(total_saving, 1)} KB')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for r in resources:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Estimate Savings From Compression: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: resources, url, app, js, size. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: resources = [| Core: for r in resources: | Evidence: print(f'total estimated saving = {round(total_saving, 1)} KB')

🛠️ Reader to Writer Bridge

1. Read: explain estimate savings from compression in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 282.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 284 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'total estimated saving = {round(total_saving, 1)} KB')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `resources = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of estimate savings from compression and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Per-resource `size * ratio` estimates the saving, and accumulating them gives the total transfer reduction you could expect from enabling compression.

```
resources = [
    {'url': 'app.js', 'size': 300, 'ratio': 0.35},
    {'url': 'style.css', 'size': 80, 'ratio': 0.25},
    {'url': 'data.json', 'size': 120, 'ratio': 0.30},
]
total_saving = 0
for r in resources:
    saved = round(r['size'] * r['ratio'], 1)
    total_saving += saved
    print(f"{r['url']}: save {saved} KB")
print(f'total estimated saving = {round(total_saving, 1)} KB')
```

```
output
app.js: save 105.0 KB
style.css: save 20.0 KB
data.json: save 36.0 KB
total estimated saving = 161.0 KB
```

How it works:

1. resources is a list of dicts each with size and a compression ratio.
2. The loop multiplies size by ratio, rounds the saving, adds it to total_saving, and prints the per-file figure.
3. After the loop it prints the rounded total of 161.0 KB saved.

Try it:

1. Sort the resources so the biggest saving prints first.
2. Skip any resource whose estimated saving is under 5 KB as not worth the effort.
3. Express the total saving as a percentage of the combined original size.
4. Raise app.js's ratio to 0.7 and watch the total saving jump.

⚠ Common Mistakes

- Treating the ratio as the compressed size rather than the fraction saved.
- Applying a text-compression ratio to already-compressed binary assets.
- Summing unrounded floats then rounding only at the end can disagree with per-row rounding.

Skills gained savings estimate | accumulator pattern | round()

→ Where You'll Use This: Estimating compression savings helps you justify the effort of configuring Brotli, turning a vague best practice into a concrete kilobyte figure for stakeholders.

Mental Model

Like estimating how much a vacuum bag will shrink each pile of laundry: a quick multiply tells you the space you will reclaim before you start sealing bags.

Project 284 Count Requests by Resource Type

The Idea Behind It

A request waterfall is far easier to reason about when grouped by type. This project feeds the type field of each request dict into a Counter via a generator, then prints the total fetch count followed by a per-type tally. Seeing two scripts against one document, stylesheet, image, and font shows where the request count concentrates.

What You'll Build & Learn

Tally HTTP requests by their resource type to show where the fetch count concentrates.

- Grouping a waterfall by type clarifies it
- Counter tallies from a generator
- Total fetches is just the list length

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 283: estimate savings from compression. Project 285 changes the surface problem while preserving the underlying method.

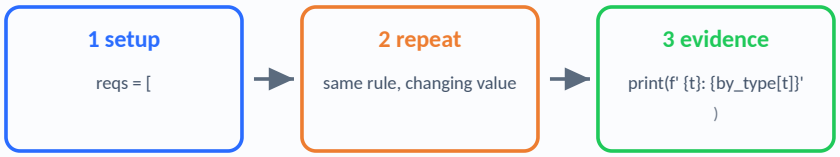
progress check

Reading target: count requests by resource type is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `reqs = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f' {t}: {by_type[t]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`reqs = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`by_type = Counter(r['type'] for r in reqs)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f' {t}: {by_type[t]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `by_type = Counter(r['type'] for r in reqs)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Requests by Resource Type: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: reqs, url, page, html, type. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: reqs = [| Core: by_type = Counter(r['type'] for r in reqs) | Evidence: print(f' {t}: {by_type[t]}')

🛠️ Reader to Writer Bridge

1. Read: explain count requests by resource type in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 283.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 285 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f' {t}: {by_type[t]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `reqs = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count requests by resource type and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A Counter over the `type` field groups the waterfall, and sorting the types makes the breakdown deterministic; too many scripts is a classic smell.

```
reqs = [
    {'url': 'page.html', 'type': 'document'},
    {'url': 'app.js', 'type': 'script'},
    {'url': 'vendor.js', 'type': 'script'},
    {'url': 'style.css', 'type': 'stylesheet'},
    {'url': 'hero.jpg', 'type': 'image'},
    {'url': 'font.woff2', 'type': 'font'},
]
```

```
from collections import Counter
by_type = Counter(r['type'] for r in reqs)
print('total fetches:', len(reqs))
for t in sorted(by_type):
    print(f'  {t}: {by_type[t]}')
```

```
output
total fetches: 6
  document: 1
  font: 1
  image: 1
  script: 2
  stylesheet: 1
```

How it works:

1. reqs is a list of six dicts, each with a url and a type.
2. Counter is fed a generator of r['type'] values, building counts per type.
3. The code prints the total of six fetches, then loops the sorted Counter to show script at two and the rest at one each.

Try it:

1. Also report which type has the highest request count using the Counter's most_common.
2. Add a size field and sum bytes per type alongside the request count.
3. Flag when scripts exceed a chosen count, hinting at too many separate bundles.
4. Add a third script and watch the script tally rise to 3.

⚠ Common Mistakes

- Counting unique URLs instead of types answers a different question.
- Forgetting that one HTML document generates many sub-resource requests.
- Printing the Counter directly leaves the type order nondeterministic.

Skills gained Counter | request grouping | generator expression

→ Where You'll Use This: Grouping requests by type is the first cut of any waterfall analysis, helping you decide whether to bundle scripts, sprite images, or reduce font variants.

Mental Model

Like sorting a pile of mail into bills, ads, and letters: once the stacks are separate, you immediately see which kind is flooding your mailbox.

Project 285 Measure Critical Request Chain Depth

The Idea Behind It

The critical request chain is the dependency path from the HTML to the last blocking resource, and its depth caps how quickly rendering can begin. This project models the chain as a dict mapping each resource to the next it depends on, then walks from the start, following links until it hits None. Tracking the order and counting hops yields a chain depth of two.

What You'll Build & Learn

Walk a dependency chain from the HTML to the final resource and report its order and depth.

- The critical chain caps render start
- A dict can model a dependency path
- Walk until the next link is None

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 284: count requests by resource type. Project 286 changes the surface problem while preserving the underlying method.

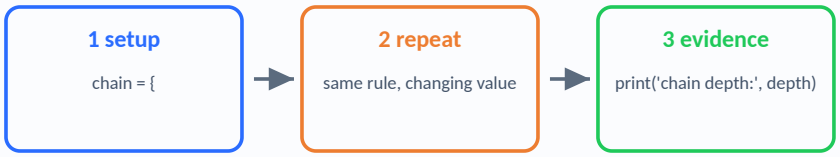
progress check

Reading target: measure critical request chain depth is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `chain = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('chain depth:', depth)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`chain = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`while chain.get(cur):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('chain depth:', depth)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `while chain.get(cur):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure Critical Request Chain Depth: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: chain, page, html, style, css. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: chain = { | Core: while chain.get(cur): | Evidence: print('chain depth:', depth)

🛠️ Reader to Writer Bridge

1. Read: explain measure critical request chain depth in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 284.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 286 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('chain depth:', depth)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `chain = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure critical request chain depth and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Walking the chain map until a resource has no dependency counts the hops; a deep chain (HTML waits on CSS waits on a font) serialises the critical path.

```
chain = {
    'page.html': 'style.css',
    'style.css': 'font.woff2',
    'font.woff2': None,
}
start = 'page.html'
depth, cur = 0, start
```

```
order = [start]
while chain.get(cur):
    cur = chain[cur]
    order.append(cur)
    depth += 1
print('critical chain:', ' -> '.join(order))
print('chain depth:', depth)
```

```
output
critical chain: page.html -> style.css -> font.woff2
chain depth: 2
```

How it works:

1. A chain dict maps page.html to style.css, style.css to font.woff2, and font.woff2 to None.
2. Starting at page.html, a while loop follows chain.get(cur), appending each next link and incrementing depth.
3. It prints the arrow-joined order and a depth of two, since two hops separate the HTML from the font.

Try it:

1. Add a fourth link to the chain and confirm the depth count rises accordingly.
2. Detect a cycle by tracking visited nodes so a self-referential chain cannot loop forever.
3. Return the order list from a function so other code can inspect the path.
4. Add a fourth link in the chain and watch the depth grow.

⚠ Common Mistakes

- Counting nodes instead of edges overstates the depth by one.
- No guard against a cyclic dependency would loop forever.
- Using `chain[cur]` instead of `chain.get(cur)` raises on the terminal node.

Skills gained chain traversal | while loop | dependency mapping

→ Where You'll Use This: Lighthouse reports critical request chains, and shortening the longest one, by preloading or removing a dependency, is a direct lever on how soon a page can render.

⚡ Mental Model

Like a relay race where each runner must finish before the next starts: the team can only be as quick as the full baton hand-off sequence allows.

Project 286 Compare Mobile and Desktop Vitals

💡 The Idea Behind It

Google indexes mobile-first, and mobile vitals are almost always worse thanks to slower CPUs and networks. This project pairs a mobile and a desktop metric dict on the same keys and, per metric, names which form factor is slower using a simple comparison. With mobile losing on all three, the output quantifies the mobile performance penalty.

What You'll Build & Learn

Compare mobile and desktop vitals per metric and name which form factor is slower each time.

- Google ranks on mobile-first indexing
- Mobile metrics are usually worse
- A comparison picks the slower side

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 285: measure critical request chain depth. Project 287 changes the surface problem while preserving the underlying method.

progress check

Reading target: compare mobile and desktop vitals is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

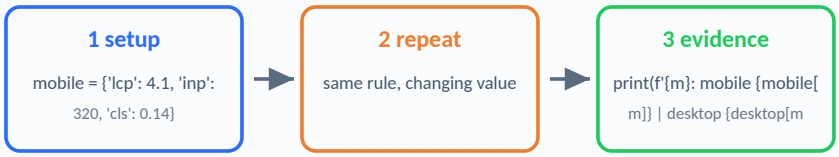
- Setup: `mobile = {'lcp': 4.1, 'inp': 320, 'cls': 0.14}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{m}: mobile {mobile[m]} | desktop {desktop[m]} -> {worse} slower')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`mobile = {'lcp': 4.1, 'inp': 320, 'cls': 0.14}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`worse = 'mobile' if mobile[m] > desktop[m] else 'desktop'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{m}: mobile {mobile[m]} | desktop {desktop[m]} -> {worse} slower')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `worse = 'mobile' if mobile[m] > desktop[m] else 'desktop'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compare Mobile and Desktop Vitals: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: mobile, lcp, inp, cls, desktop. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: mobile = {'lcp': 4.1, 'inp': 320, 'cls': 0.14} | Core: worse = 'mobile' if mobile[m] > desktop[m] else 'desktop' | Evidence: print(f'{m}: mobile {mobile[m]} | desktop {desktop[m]} -> {worse} slowe...

Reader to Writer Bridge

1. Read: explain compare mobile and desktop vitals in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 285.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 287 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{m}: mobile {mobile[m]} | desktop {desktop[m]} -> {worse} slower')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `mobile = {'lcp': 4.1, 'inp': 320, 'cls': 0.14}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compare mobile and desktop vitals and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Comparing each metric across the two device dicts names the slower form factor; mobile losing on every metric is the typical and concerning pattern.

```
mobile = {'lcp': 4.1, 'inp': 320, 'cls': 0.14}
desktop = {'lcp': 1.8, 'inp': 90, 'cls': 0.03}
for m in sorted(mobile):
    worse = 'mobile' if mobile[m] > desktop[m] else 'desktop'
    print(f'{m}: mobile {mobile[m]} | desktop {desktop[m]} -> {worse} slower')
```

```
output
cls: mobile 0.14 | desktop 0.03 -> mobile slower
inp: mobile 320 | desktop 90 -> mobile slower
lcp: mobile 4.1 | desktop 1.8 -> mobile slower
```

How it works:

1. Two dicts, mobile and desktop, carry lcp, inp, and cls readings.
2. For each sorted metric a conditional picks mobile when its value is higher, else desktop, as slower.
3. The output shows mobile slower on all three, with lcp 4.1 against desktop 1.8.

Try it:

1. Print the size of each gap, not just which device is slower.
2. Flag any metric where desktop is actually the slower one, which would be unusual.
3. Grade both readings into bands and show how often mobile drops a whole band below desktop.
4. Make desktop CLS worse than mobile and watch one row flip.

⚠ Common Mistakes

- Comparing only the overall pass hides which metric is worse on which device.
- Assuming higher is always worse; it is for these latency/shift metrics but not for scores.
- Iterating without sorting keys yields nondeterministic metric order.

Skills gained mobile-first | per-metric comparison | dict pairing

→ Where You'll Use This: Because rankings hinge on the mobile experience, this side-by-side is how you justify prioritising mobile fixes even when your desktop scores already look healthy.

⚡ Mental Model

Like timing the same recipe on a camp stove versus a full kitchen range: the underpowered burner finishes later on nearly every dish.

Project 287 Compute the p75 of LCP Samples

💡 The Idea Behind It

CrUX reports the 75th percentile rather than the average, so a handful of fast visits cannot hide many slow ones. This project sorts the LCP samples, computes a fractional rank at 0.75 of the last index, and linearly interpolates between the two bracketing values. Reproducing this p75 of 3.4s mirrors how the field metric is actually derived.

What You'll Build & Learn

Interpolate the 75th percentile of a list of LCP samples to reproduce the CrUX field metric.

- CrUX uses the 75th percentile
- Percentiles resist a few fast outliers
- Linear interpolation between ranks

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 286: compare mobile and desktop vitals. Project 288 changes the surface problem while preserving the underlying method.

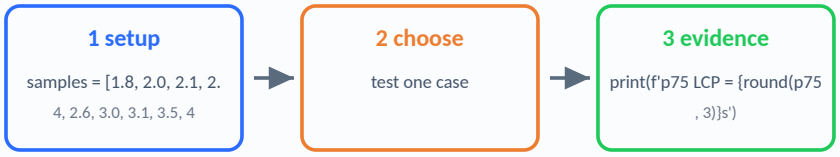
progress check

Reading target: compute the p75 of lcp samples is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `samples = [1.8, 2.0, 2.1, 2.4, 2.6, 3.0, 3.1, 3.5, 4.0, 4.2]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'p75 LCP = {round(p75, 3)}s')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`samples = [1.8, 2.0, 2.1, 2.4, 2.6, 3.0, 3.1, 3.5, 4.0, 4.2]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.

Move	Code to inspect	Why it matters
2. Apply the rule	`p75 = ordered[lo] + frac * (ordered[lo + 1] - ordered[lo]) if frac else ordered[lo]`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(f'p75 LCP = {round(p75, 3)}s')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `p75 = ordered[lo] + frac * (ordered[lo + 1] - ordered[lo]) if frac else ordered[lo]`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Compute the p75 of LCP Samples: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: samples, ordered, sorted, rank, len. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: samples = [1.8, 2.0, 2.1, 2.4, 2.6, 3.0, 3.1, 3.5, 4.0, 4.2] | Core: p75 = ordered[lo] + frac * (ordered[lo + 1] - ordered[lo]) if frac else... | Evidence: print(f'p75 LCP = {round(p75, 3)}s')

🛠️ Reader to Writer Bridge

1. Read: explain compute the p75 of lcp samples in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 286.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 288 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'p75 LCP = {round(p75, 3)}s')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `samples = [1.8, 2.0, 2.1, 2.4, 2.6, 3.0, 3.1, 3.5, 4.0, 4.2]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute the p75 of lcp samples and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Sorting the samples and linearly interpolating at rank `0.75 * (n-1)` gives the 75th percentile, the exact statistic Google grades Core Web Vitals against.

```
samples = [1.8, 2.0, 2.1, 2.4, 2.6, 3.0, 3.1, 3.5, 4.0, 4.2]
ordered = sorted(samples)
rank = 0.75 * (len(ordered) - 1)
lo = int(rank)
frac = rank - lo
p75 = ordered[lo] + frac * (ordered[lo + 1] - ordered[lo]) if frac else
ordered[lo]
print(f'p75 LCP = {round(p75, 3)}s')
```

```
output
p75 LCP = 3.4s
```

How it works:

1. The samples list is sorted, and rank is 0.75 times the last index, giving a fractional position.
2. lo is the integer part and frac the remainder used to interpolate between ordered[lo] and the next value.
3. Since frac is non-zero, the interpolation yields a p75 LCP printed as 3.4 seconds.

Try it:

1. Parameterise the percentile so the same code can compute p90 or the median.
2. Handle the edge case of a single-sample list without an index error.
3. Grade the resulting p75 into the LCP bands to give it a verdict.

4. Compute the p90 by changing 0.75 to 0.90.

⚠ Common Mistakes

- Reporting the mean instead of the p75 hides the slow tail Google penalises.
- Off-by-one in the rank index (`n` vs `n-1`) skews the percentile.
- Skipping the sort means the percentile is computed over unordered data.

Skills gained percentile interpolation | sorting | field metrics

→ Where You'll Use This: Knowing p75 is the reporting basis explains why Google cares about your worst quartile of visitors, shaping how you read CrUX dashboards and set targets.

⚡ Mental Model

Like a school reporting the mark three-quarters of students beat rather than the class average: it deliberately exposes the trailing group instead of hiding it.

Project 288 Detect a Budget Regression vs a Baseline

💡 The Idea Behind It

A budget that passes today can quietly bloat over time, so this project diffs current per-type sizes against a stored baseline. It loops the shared keys, computes each difference, records only the positive growths in a regressions dict, and prints a signed change per type. The result names image and js as the categories that grew since the baseline.

What You'll Build & Learn

Diff current per-type sizes against a baseline and flag the resource types that regressed upward.

- Budgets bloat without a baseline
- Positive diffs are regressions
- Record growth in a separate dict

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 287: compute the p75 of lcp samples. Project 289 changes the surface problem while preserving the underlying method.

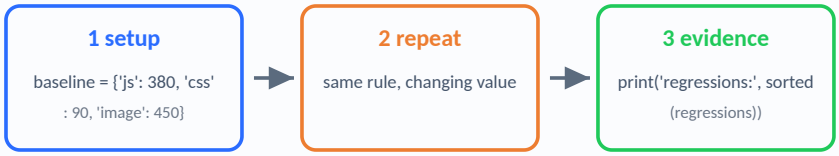
progress check

Reading target: detect a budget regression vs a baseline is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `baseline = {'js': 380, 'css': 90, 'image': 450}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('regressions:', sorted(regressions))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`baseline = {'js': 380, 'css': 90, 'image': 450}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`if diff > 0:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('regressions:', sorted(regressions))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if diff > 0:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Budget Regression vs a Baseline: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: baseline, js, css, image, current. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: baseline = {'js': 380, 'css': 90, 'image': 450} | Core: if diff > 0: | Evidence: print('regressions:', sorted(regressions))

🛠️ Reader to Writer Bridge

1. Read: explain detect a budget regression vs a baseline in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 287.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 289 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('regressions:', sorted(regressions))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `baseline = {'js': 380, 'css': 90, 'image': 450}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of detect a budget regression vs a baseline and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Subtracting baseline from current per type signs each change, and collecting the positive diffs lists exactly which resource types regressed since the baseline.

```
baseline = {'js': 380, 'css': 90, 'image': 450}
current = {'js': 460, 'css': 85, 'image': 470}
regressions = {}
for k in sorted(baseline):
    diff = current[k] - baseline[k]
    if diff > 0:
        regressions[k] = diff
    print(f'{k}: {baseline[k]} -> {current[k]} ({"+" if diff >= 0 else ""}
{diff} KB)')
print('regressions:', sorted(regressions))
```

```
output
css: 90 -> 85 (-5 KB)
image: 450 -> 470 (+20 KB)
js: 380 -> 460 (+80 KB)
regressions: ['image', 'js']
```

How it works:

1. baseline and current dicts share keys js, css, and image.
2. For each sorted key the diff is current minus baseline; positive diffs are stored in regressions and every change prints signed.
3. css shrank by 5 while image grew 20 and js grew 80, so regressions lists image and js.

Try it:

1. Only count a regression when growth exceeds a tolerance, ignoring trivial jitter.
2. Print the total kilobytes gained across all regressed types.
3. Track improvements too and report which types shrank the most.
4. Make every type shrink and confirm zero regressions are reported.

⚠ Common Mistakes

- Flagging any change as a regression, including the CSS that actually shrank.
- Comparing totals instead of per type hides a JS gain offset by an image loss.

- Missing a key present in one dict but not the other raises a KeyError.

Skills gained regression detection baseline diffing signed change

→ Where You'll Use This: Comparing against a committed baseline is how CI catches creeping page-weight regressions, failing a pull request that quietly adds a heavy script or image.

Mental Model

Like weighing yourself against last month's number: the scale only flags the pounds that crept on, telling you exactly where the drift happened.

Project 289 Track a Performance Score Trend Across Builds

The Idea Behind It

CI can capture a Lighthouse score per build, and walking those scores in build order turns a column of numbers into a readable trend. This project iterates a sorted builds dict, comparing each score to the previous to print an up, down, or flat marker, then reports the net change from first to last. The build-101 to build-104 swing surfaces a noisy but net-positive trajectory.

What You'll Build & Learn

Walk per-build performance scores in order, mark each up, down, or flat, and report the net change.

- CI can log a score per build
- Compare each value to the previous
- Net change is last minus first

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 288: detect a budget regression vs a baseline. Project 290 changes the surface problem while preserving the underlying method.

progress check

Reading target: track a performance score trend across builds is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

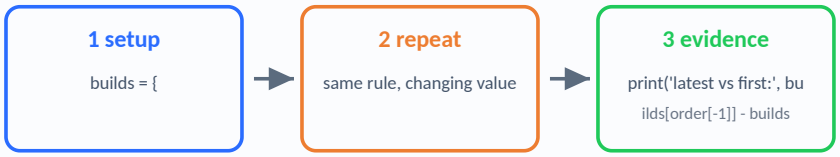
Logic Lens

- Setup: `builds = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('latest vs first:', builds[order[-1]] - builds[order[0]])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`builds = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`arrow = '' if prev is None else (' up' if score > prev else ' down' if score < prev else ' flat')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('latest vs first:', builds[order[-1]] - builds[order[0]])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
arrow = '' if prev is None else (' up' if score > prev else ' down' if score < prev else ' flat')
```

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Track a Performance Score Trend Across Builds: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: builds, build, order, sorted, prev. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: builds = { | Core: arrow = '' if prev is None else (' up' if score > prev else ' down' if... | Evidence: print('latest vs first:', builds[order[-1]] - builds[order[0]])

🛠️ Reader to Writer Bridge

1. Read: explain track a performance score trend across builds in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 288.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 290 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('latest vs first:', builds[order[-1]] - builds[order[0]])`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `builds = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of track a performance score trend across builds and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Iterating the sorted build ids and comparing each score to the previous one annotates the direction of change, and the first-to-last delta summarises the overall trend.

```
builds = {
    'build-101': 78,
    'build-102': 82,
    'build-103': 71,
    'build-104': 88,
}
order = sorted(builds)
prev = None
for b in order:
    score = builds[b]
    arrow = '' if prev is None else (' up' if score > prev else ' down' if
score < prev else ' flat')
    print(f'{b}: {score}{arrow}')
    prev = score
print('latest vs first:', builds[order[-1]] - builds[order[0]])
```

```
output
build-101: 78
build-102: 82 up
build-103: 71 down
build-104: 88 up
latest vs first: 10
```

How it works:

1. A builds dict maps build labels to scores, iterated in sorted order.
2. A prev variable holds the last score; each iteration appends an up, down, or flat arrow based on the comparison.
3. After the loop it prints latest minus first, showing a net gain of 10 from build-101 to build-104.

Try it:

1. Print the per-step delta number alongside each arrow.
2. Find and flag the single build with the steepest drop.
3. Compute the average score across all builds as a stability summary.
4. Insert a regressing build between 102 and 103 and watch a 'down' appear.

⚠ Common Mistakes

- Iterating an unsorted dict scrambles the build order and the trend.
- Comparing each build to the first instead of the previous mislabels the arrows.
- Treating the very first build as 'down' instead of having no prior to compare.

Skills gained trend marking | previous-value compare | net change

→ Where You'll Use This: Tracking a Lighthouse score across builds is how teams spot the commit that regressed performance, making this walk the core of a CI trend report.

⚡ Mental Model

Like reading a fitness app's daily steps: each day shows an arrow versus yesterday, and the month-end figure tells you whether you really moved forward.

Project 290 Compute Total Blocking Time From Tasks

💡 The Idea Behind It

Total Blocking Time sums how long main-thread tasks ran past 50ms, since only that excess slice actually blocks user input. This project filters the task list to long tasks over the threshold, then sums each task's time minus 50 to get TBT. The resulting 250ms is the lab proxy that predicts poor INP in the field.

What You'll Build & Learn

Compute Total Blocking Time by summing each long task's milliseconds past the 50ms threshold.

- TBT counts only time past 50ms
- Long tasks block user input
- TBT is the lab proxy for INP

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 289: track a performance score trend across builds. Project 291 changes the surface problem while preserving the underlying method.

progress check

Reading target: compute total blocking time from tasks is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `tasks = [50, 120, 30, 80, 200, 45]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'total blocking time = {tbt}ms')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`tasks = [50, 120, 30, 80, 200, 45]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`long_tasks = [t for t in tasks if t > threshold]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'total blocking time = {tbt}ms')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `long_tasks = [t for t in tasks if t > threshold]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute Total Blocking Time From Tasks: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: tasks, threshold, long_tasks, tbt, sum. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: tasks = [50, 120, 30, 80, 200, 45] | Core: long_tasks = [t for t in tasks if t > threshold] | Evidence: print(f'total blocking time = {tbt}ms')

🛠️ Reader to Writer Bridge

1. Read: explain compute total blocking time from tasks in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 289.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 291 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'total blocking time = {tbt}ms')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tasks = [50, 120, 30, 80, 200, 45]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of compute total blocking time from tasks and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

For each task over 50ms only the excess counts toward TBT, so summing `duration - 50` over the long tasks reproduces the metric that predicts responsiveness.

```
tasks = [50, 120, 30, 80, 200, 45]
threshold = 50
long_tasks = [t for t in tasks if t > threshold]
tbt = sum(t - threshold for t in long_tasks)
print(f'long tasks (> {threshold}ms): {len(long_tasks)}')
print(f'total blocking time = {tbt}ms')
```

```
output
long tasks (> 50ms): 3
total blocking time = 250ms
```

How it works:

1. tasks is a list of durations and threshold is 50.
2. A comprehension keeps tasks over 50, then a generator sums each one's time minus 50.
3. It prints three long tasks and a total blocking time of 250 milliseconds.

Try it:

1. Report which single task contributed the most blocking time.
2. Make the 50ms threshold a parameter so you can experiment with stricter cutoffs.
3. Count how many tasks fall under the threshold and are therefore harmless.
4. Add a 60ms task and watch TBT rise by 10.

⚠ Common Mistakes

- Summing whole task durations instead of just the portion above 50ms.
- Including tasks at exactly 50ms, which contribute zero blocking time.
- Hard-coding 50 in two spots so the threshold and message can disagree.

Skills gained TBT calculation | threshold filter | generator sum

→ Where You'll Use This: Because TBT in the lab predicts INP in the field, reducing it, by splitting long JavaScript tasks, is a concrete way to improve real-user responsiveness.

Mental Model

Like measuring only the minutes a meeting runs past its scheduled end: short ones cost nothing, and it is the overruns that eat everyone's afternoon.

Project 291 Compute CLS From Layout Shift Records

The Idea Behind It

CLS is the sum of impact-fraction times distance-fraction across every unexpected layout shift on a page. This project builds the score directly from a list of shift records, multiplying each record's two fields and summing the products with a generator. Reconstructing CLS as 0.09 and grading it good shows exactly where the headline score comes from.

What You'll Build & Learn

Build a CLS score from layout-shift records by summing impact times distance, then grade it.

- CLS sums impact times distance
- Each shift contributes its own product
- Grade the sum against the bands

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 290: compute total blocking time from tasks. Project 292 changes the surface problem while preserving the underlying method.

progress check

Reading target: compute cls from layout shift records is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

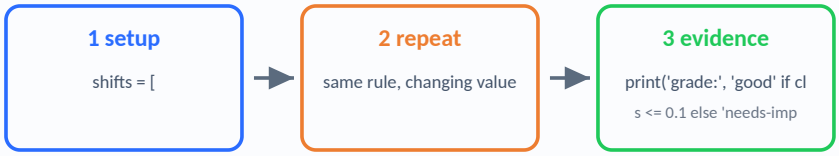
- Setup: `shifts = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <= 0.25 else 'poor')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`shifts = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <= 0.25 else 'poor')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <= 0.25 else 'poor')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <= 0.25 else 'poor')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute CLS From Layout Shift Records: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: shifts, impact, distance, cls, sum. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: shifts = [| Core: print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <=... | Evidence: print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <=...

🛠️ Reader to Writer Bridge

1. Read: explain compute cls from layout shift records in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 290.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 292 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <= 0.25 else 'poor')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `shifts = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute cls from layout shift records and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each shift contributes `impact * distance`, and summing them yields the CLS that the banded grade then classifies.

```
shifts = [
    {'impact': 0.15, 'distance': 0.3},
    {'impact': 0.25, 'distance': 0.1},
    {'impact': 0.10, 'distance': 0.2},
]
cls = sum(s['impact'] * s['distance'] for s in shifts)
print(f'CLS = {round(cls, 4)}')
print('grade:', 'good' if cls <= 0.1 else 'needs-improvement' if cls <= 0.25
else 'poor')
```

```
output
CLS = 0.09
grade: good
```

How it works:

1. shifts is a list of dicts, each with an impact and a distance fraction.
2. A generator multiplies impact by distance per record and sum() combines them into 0.09.
3. A chained conditional grades the rounded score good because it stays at or under 0.1.

Try it:

1. Print the single shift that contributed the most to the score.
2. Add a shift large enough to push the total into needs-improvement and confirm the grade changes.
3. Round the per-shift contributions and show each one's percentage of the total.
4. Add a fourth large shift and watch CLS cross into needs-improvement.

⚠ Common Mistakes

- Adding impact and distance instead of multiplying them inflates the score.
- Counting expected (user-initiated) shifts that should be excluded from CLS.
- Printing the unrounded float gives a long, unstable decimal.

Skills gained CLS computation | generator sum | band grading

→ Where You'll Use This: Understanding CLS as a sum of shift products helps you trace a bad score to the specific elements moving, like a late-loading banner shoving content down.

Mental Model

Like scoring a clumsy dance routine by how big and how far each stumble was: many small trips add up to the same mark as one dramatic lurch.

Project 292 Flag Fonts Risking Invisible Text

The Idea Behind It

Without font-display: swap a web font can hide text entirely while it downloads, the flash of invisible text that delays LCP. This project scans a list of font dicts and collects any whose display value is not exactly swap, including block and None. Listing Roboto and Lora flags the declarations that risk invisible text on slow connections.

What You'll Build & Learn

List web fonts lacking font-display: swap, which risk a flash of invisible text and slow LCP.

- font-display swap avoids invisible text
- FOIT can delay LCP
- Filter fonts not set to swap

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 291: compute cls from layout shift records. Project 293 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag fonts risking invisible text is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

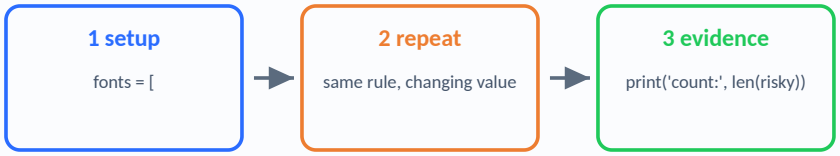
- Setup: `fonts = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('count:', len(risky))` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`fonts = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`risky = [f['family'] for f in fonts if f['display'] != 'swap']`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('count:', len(risky))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `risky = [f['family'] for f in fonts if f['display'] != 'swap']`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Fonts Risking Invisible Text: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: fonts, family, Inter, display, swap. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: fonts = [| Core: risky = [f['family'] for f in fonts if f['display'] != 'swap'] | Evidence: print('count:', len(risky))

Reader to Writer Bridge

1. Read: explain flag fonts risking invisible text in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 291.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 293 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print('count:', len(risky))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `fonts = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag fonts risking invisible text and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Filtering fonts whose `display` value is not 'swap' catches both 'block' and missing values, the two settings that can delay text rendering.

```
fonts = [
    {'family': 'Inter', 'display': 'swap'},
    {'family': 'Roboto', 'display': 'block'},
    {'family': 'Lora', 'display': None},
]
risky = [f['family'] for f in fonts if f['display'] != 'swap']
```

```
print('fonts risking invisible text:', sorted(risky))
print('count:', len(risky))
```

```
output
fonts risking invisible text: ['Lora', 'Roboto']
count: 2
```

How it works:

1. fonts is a list of dicts with a family and a display value of swap, block, or None.
2. A comprehension keeps the family where display is not equal to swap, catching Roboto and Lora.
3. It prints the sorted risky families and a count of two.

Try it:

1. Treat a display of optional as acceptable too, since it also avoids long invisible-text periods.
2. Report the total font count alongside the number at risk.
3. Add a fallback-font field and only flag fonts that lack both swap and a system fallback.
4. Set Roboto to 'swap' and watch the risky count fall to 1.

⚠ Common Mistakes

- Treating a missing `display` value as safe when the default can block rendering.
- Accepting 'optional' or 'fallback' as equivalent to 'swap' without deciding policy.
- Printing the offender set unsorted gives nondeterministic order.

Skills gained font-display audit | inequality filter | FOIT risk

→ Where You'll Use This: Auditing font-display is a routine LCP fix, since switching declarations to swap keeps text visible during font loads and is a quick rendering win.

⚡ Mental Model

Like a theatre that keeps the stage dark until the star arrives: swap turns on the work lights so the audience can at least read the program meanwhile.

Project 293 Pinpoint Which Vital To Fix First

💡 The Idea Behind It

When a page fails Core Web Vitals you need to know where to spend effort first. This project compares each metric to its threshold, collects the failing keys with a comprehension, and names the first one as the focus. Because only lcp exceeds its cap here, the output points work squarely at fixing LCP.

What You'll Build & Learn

List the metrics that exceed their thresholds and name the first failing one as the focus.

- Failing metrics guide where to work
- Comprehension collects the over-limit keys
- Index zero names the first focus

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 292: flag fonts risking invisible text. Project 294 changes the surface problem while preserving the underlying method.

progress check

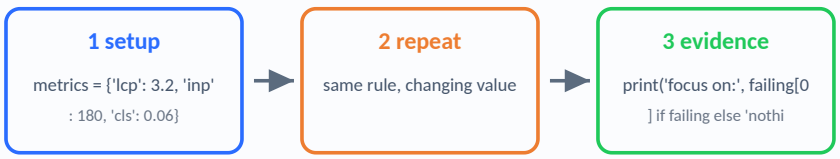
Reading target: pinpoint which vital to fix first is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `metrics = {'lcp': 3.2, 'inp': 180, 'cls': 0.06}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('focus on:', failing[0] if failing else 'nothing, all good')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`metrics = {'lcp': 3.2, 'inp': 180, 'cls': 0.06}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`failing = [m for m in thresholds if metrics[m] > thresholds[m]]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('focus on:', failing[0] if failing else 'nothing, all good')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `failing = [m for m in thresholds if metrics[m] > thresholds[m]]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Pinpoint Which Vital To Fix First: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: metrics, lcp, inp, cls, thresholds. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: metrics = {'lcp': 3.2, 'inp': 180, 'cls': 0.06} | Core: failing = [m for m in thresholds if metrics[m] > thresholds[m]] | Evidence: print('focus on:', failing[0] if failing else 'nothing, all good')

Reader to Writer Bridge

1. Read: explain pinpoint which vital to fix first in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 292.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 294 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print('focus on:', failing[0] if failing else 'nothing, all good')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `metrics = {'lcp': 3.2, 'inp': 180, 'cls': 0.06}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of pinpoint which vital to fix first and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

From Concept to Code (Python)

Filtering the metrics that exceed their thresholds yields the failure list, and the first entry (in sorted order) becomes the recommended focus.

```
metrics = {'lcp': 3.2, 'inp': 180, 'cls': 0.06}
thresholds = {'lcp': 2.5, 'inp': 200, 'cls': 0.1}
failing = [m for m in thresholds if metrics[m] > thresholds[m]]
print('failing metrics:', sorted(failing))
print('focus on:', failing[0] if failing else 'nothing, all good')
```

```
output
failing metrics: ['lcp']
focus on: lcp
```

How it works:

1. metrics and a parallel thresholds dict share lcp, inp, and cls.
2. A comprehension keeps each metric whose value exceeds its threshold, capturing only lcp.
3. It prints the sorted failing list and names failing[0] as the focus, or a clean message when nothing fails.

Try it:

1. Order the failing metrics by how far each overshoots its threshold and focus on the biggest gap.
2. Handle the all-good case without indexing into an empty list.

3. Print a tailored fix hint for whichever metric is named as the focus.
4. Make `inp` also fail and confirm both appear in the list.

⚠ Common Mistakes

- Reporting all metrics rather than only the ones over threshold.
- Indexing `failing[0]` without first checking the list is non-empty raises on a passing page.
- Not sorting the failing list makes the recommended focus nondeterministic.

Skills gained failure prioritisation | threshold comparison | list indexing

→ Where You'll Use This: Naming the worst offending metric is how you turn a red Core Web Vitals report into a single actionable ticket rather than a vague directive to be faster.

Mental Model

Like a doctor reading a panel of test results and circling the one out-of-range number: that is where treatment starts, not the values already healthy.

Project 294 Break LCP Into Its Four Sub-Parts

The Idea Behind It

LCP decomposes into four sub-parts: TTFB, resource load delay, load time, and render delay, and splitting it reveals whether the server, network, or rendering is to blame. This project sums the breakdown to the total LCP, finds the largest component with max over a key, and prints each part's share. With load_time at 42 percent it names the dominant contributor.

What You'll Build & Learn

Split an LCP value into its four sub-parts, show each share, and name the biggest contributor.

- LCP has four diagnostic sub-parts
- Shares localise the bottleneck
- max with a key finds the largest part

Skill Roadmap

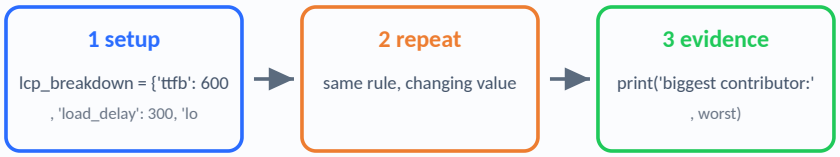
Skill step: pushes the same idea into a less automatic situation. It builds on Project 293: pinpoint which vital to fix first. Project 295 changes the surface problem while preserving the underlying method. *progress check*

Reading target: break lcp into its four sub-parts is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `lcp_breakdown = {'ttfb': 600, 'load_delay': 300, 'load_time': 800, 'render_delay': 200}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('biggest contributor:', worst)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`lcp_breakdown = {'ttfb': 600, 'load_delay': 300, 'load_time': 800, 'render_delay': 200}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`for part in sorted(lcp_breakdown):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('biggest contributor:', worst)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for part in sorted(lcp_breakdown):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Break LCP Into Its Four Sub-Parts: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: lcp_breakdown, ttfb, load_delay, load_time, render_delay. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: lcp_breakdown = {'ttfb': 600, 'load_delay': 300, 'load_time': 800, 'ren... | Core: for part in sorted(lcp_breakdown): | Evidence: print('biggest contributor:', worst)

🛠️ Reader to Writer Bridge

1. Read: explain break lcp into its four sub-parts in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 293.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 295 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('biggest contributor:', worst)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `lcp_breakdown = {'ttfb': 600, 'load_delay': 300, 'load_time': 800, 'render_delay': 200}` and say why this line, value, import, or entry point

is needed before the rest of the Python can work.

3. Transfer: change one small part of break lcp into its four sub-parts and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Summing the four sub-parts reconstructs total LCP, and turning each into a percentage of the whole names the dominant phase to optimise.

```
lcp_breakdown = {'ttfb': 600, 'load_delay': 300, 'load_time': 800,
'render_delay': 200}
total = sum(lcp_breakdown.values())
worst = max(sorted(lcp_breakdown), key=lambda k: lcp_breakdown[k])
print(f'LCP = {total}ms ({round(total / 1000, 2)}s)')
for part in sorted(lcp_breakdown):
    pct = round(100 * lcp_breakdown[part] / total)
    print(f'  {part}: {lcp_breakdown[part]}ms ({pct}%)')
print('biggest contributor:', worst)
```

```
output
LCP = 1900ms (1.9s)
  load_delay: 300ms (16%)
  load_time: 800ms (42%)
  render_delay: 200ms (11%)
  ttfb: 600ms (32%)
biggest contributor: load_time
```

How it works:

1. An lcp_breakdown dict holds ttfb, load_delay, load_time, and render_delay in milliseconds.
2. sum() gives the 1900ms total, and max over the keys by value picks load_time as the worst.
3. A loop prints each part's milliseconds and rounded percentage before naming load_time as the biggest contributor.

Try it:

1. Suggest a fix tailored to whichever sub-part dominates, like preloading when load_delay wins.
2. Flag when two sub-parts are within a few percent of each other, signalling no single culprit.
3. Convert the breakdown into a small text bar chart of the shares.
4. Triple the TTFB and watch it become the biggest contributor.

⚠ Common Mistakes

- Computing percentages against a hard-coded 2500ms instead of the actual total.
- Picking the biggest part with `max(dict)` returns a key by name, not by value.

- Dividing integer milliseconds before multiplying by 100 floors the percentage.

Skills gained LCP breakdown | percentage shares | max with key

→ **Where You'll Use This:** Breaking down LCP is how performance engineers decide whether to tune the backend, preload the resource, or simplify rendering, rather than guessing at the cause.

Mental Model

Like a courier's delivery timeline split into pickup, transit, and doorstep wait: only by stage-timing the route do you learn where the package really sat.

Project 295 Score a Blocking Penalty Across Resources

The Idea Behind It

Render-blocking resources hurt more than deferred ones, so this project weights each resource type and doubles its penalty when it blocks. It loops a manifest of (url, type, blocking) tuples, looks up the type's weight, multiplies by two for blocking resources, and accumulates a single risk score. The 2.9 result compresses a whole manifest into one comparable number.

What You'll Build & Learn

Turn a resource manifest into one risk score by weighting types and doubling the blocking ones.

- Blocking resources deserve a heavier penalty
- Per-type weights set base severity
- An accumulator sums into one score

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 294: break lcp into its four sub-parts. Project 296 changes the surface problem while preserving the underlying method.

progress check

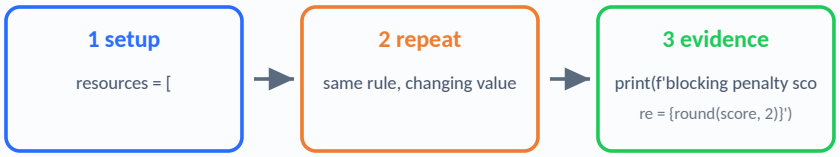
Reading target: score a blocking penalty across resources is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `resources = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'blocking penalty score = {round(score, 2)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`resources = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`penalty = weights[kind] * (2 if blocking else 1)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'blocking penalty score = {round(score, 2)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `penalty = weights[kind] * (2 if blocking else 1)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score a Blocking Penalty Across Resources: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: resources, app, js, hero, jpg. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: resources = [| Core: penalty = weights[kind] * (2 if blocking else 1) | Evidence: print(f'blocking penalty score = {round(score, 2)}')

🛠️ Reader to Writer Bridge

1. Read: explain score a blocking penalty across resources in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 294.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 296 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'blocking penalty score = {round(score, 2)}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `resources = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score a blocking penalty across resources and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each resource adds its type weight, doubled when it blocks rendering, so the accumulated score rewards moving heavy critical resources off the critical path.

```
resources = [
    ('app.js', 'js', True),
    ('hero.jpg', 'image', False),
    ('style.css', 'css', True),
    ('analytics.js', 'js', False),
    ('font.woff2', 'font', True),
]
weights = {'js': 0.3, 'image': 0.2, 'css': 0.4, 'font': 0.5}
score = 0.0
for url, kind, blocking in resources:
    penalty = weights[kind] * (2 if blocking else 1)
    score += penalty
print(f'blocking penalty score = {round(score, 2)}')
```

```
output
blocking penalty score = 2.9
```

How it works:

1. resources is a list of tuples with url, type, and a blocking boolean, plus a weights dict per type.
2. The loop looks up each type's weight and multiplies by two when blocking, adding the penalty to score.
3. After the loop it prints the rounded blocking penalty score of 2.9.

Try it:

1. Also factor each resource's size into the penalty so heavy blockers score worse.
2. Print the per-resource penalty so the biggest contributor is visible.
3. Use a default weight so an unknown resource type does not raise a KeyError.
4. Make hero.jpg blocking and watch the penalty rise.

⚠ Common Mistakes

- Applying the doubling additively (weight + 2) instead of multiplicatively (weight * 2).
- Missing a resource type in the weights dict raises a KeyError.

- Summing into an int then printing a float-looking value without rounding.

Skills gained weighted scoring blocking penalty accumulator pattern

→ Where You'll Use This: A composite blocking score lets you rank pages or templates by render risk, turning a scattered list of resources into a single metric you can trend over time.

Mental Model

Like an insurance risk score: each factor carries a weight, and the dangerous ones, here the blockers, count double toward the final premium.

Project 296 Assess CrUX Good-Density Against 75 Percent

The Idea Behind It

In the field a metric passes only when at least 75 percent of samples fall in the good band, the CrUX good-density rule. This project sums a sample distribution across good, needs-improvement, and poor, computes the good share as a rounded percentage, and asserts PASS only above 75. With good at 72 percent the assessment is FAIL, just under the bar.

What You'll Build & Learn

Compute the good-sample density from a CrUX distribution and decide PASS or FAIL against 75 percent.

- Field pass needs 75 percent good
- Good-density comes from the distribution
- Just under the bar still fails

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 295: score a blocking penalty across resources. Project 297 changes the surface problem while preserving the underlying method.

progress check

Reading target: assess crux good-density against 75 percent is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

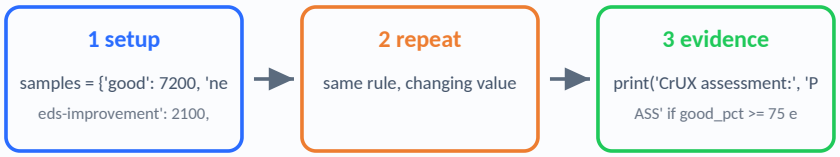
Logic Lens

- Setup: `samples = {'good': 7200, 'needs-improvement': 2100, 'poor': 700}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`samples = {'good': 7200, 'needs-improvement': 2100, 'poor': 700}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Assess CrUX Good-Density Against 75 Percent: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: samples, good, needs, improvement, poor. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: samples = {'good': 7200, 'needs-improvement': 2100, 'poor': 700} | Core: print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL') | Evidence: print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL')

🛠️ Reader to Writer Bridge

1. Read: explain assess crux good-density against 75 percent in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 295.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 297 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `samples = {'good': 7200, 'needs-improvement': 2100, 'poor': 700}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of assess crux good-density against 75 percent and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Dividing the good count by the total gives the good-density percentage, and the 75% rule decides whether the metric passes for real users.

```
samples = {'good': 7200, 'needs-improvement': 2100, 'poor': 700}
total = sum(samples.values())
good_pct = round(100 * samples['good'] / total, 1)
print('field distribution:')
for k in ['good', 'needs-improvement', 'poor']:
    print(f"  {k}: {round(100 * samples[k] / total, 1)}%")
print('CrUX assessment:', 'PASS' if good_pct >= 75 else 'FAIL')
```

```
output
field distribution:
  good: 72.0%
  needs-improvement: 21.0%
  poor: 7.0%
CrUX assessment: FAIL
```

How it works:

1. A samples dict holds counts for good, needs-improvement, and poor, totalling 10000.
2. good_pct is the good count over the total times 100, rounded to 72.0.
3. A loop prints each band's percentage, then the assessment prints FAIL because 72 is below 75.

Try it:

1. Print how many more good samples would be needed to cross 75 percent.
2. Run the rule against several metric distributions and tally how many pass.
3. Combine the good and needs-improvement shares to report an at-or-better figure too.
4. Push the good count to 7600 and watch the assessment flip to PASS.

⚠ Common Mistakes

- Comparing the good count to a raw number instead of a percentage of the total.
- Using the 'poor' bucket for the pass test instead of the 'good' bucket.
- Rounding the three buckets independently so they fail to sum to 100%.

Skills gained good-density rule percentage math CrUX assessment

→ Where You'll Use This: The 75 percent threshold is exactly how Google decides whether a metric passes in the field, so this calculation mirrors the verdict behind a CrUX report.

Mental Model

Like a vote needing a supermajority: a clear plurality is not enough, and falling three points shy of seventy-five means the motion simply does not carry.

Project 297 Quantify Before-and-After Optimisation Gains

The Idea Behind It

After an optimisation pass you want a concrete number to prove the win. This project totals the before and after per-type sizes, subtracts to find the kilobytes saved, and expresses that saving as a percentage of the original weight. The 290 KB and 22.8 percent figures turn a vague faster into a quantified result.

What You'll Build & Learn

Diff total page weight before and after optimisation and report the saving in KB and percent.

- Before-and-after proves an optimisation
- sum() totals each snapshot
- Saving as KB and as a percentage

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 296: assess crux good-density against 75 percent. Project 298 changes the surface problem while preserving the underlying method.

progress check

Reading target: quantify before-and-after optimisation gains is the surface task; the transferable Python move is to mark setup, rule, and output before running.

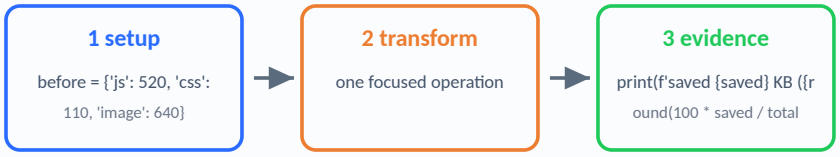
setup -> rule -> result

Logic Lens

- Setup: `before = {'js': 520, 'css': 110, 'image': 640}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is deliberately small: Python stores a value, applies one clear operation, and prints evidence. That tiny chain is the reading pattern larger programs still use.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'saved {saved} KB ({round(100 * saved / total_before, 1)}%)')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`before = {'js': 520, 'css': 110, 'image': 640}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`saved = total_before - total_after`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. slow line
3. Check the evidence	`print(f'saved {saved} KB ({round(100 * saved / total_before, 1)}%)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `saved = total_before - total_after`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Quantify Before-and-After Optimisation Gains: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: before, js, css, image, after. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: before = {'js': 520, 'css': 110, 'image': 640} | Core: saved = total_before - total_after | Evidence: print(f'saved {saved} KB ({round(100 * saved / total_before, 1)}%)')

🛠️ Reader to Writer Bridge

1. Read: explain quantify before-and-after optimisation gains in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 296.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 298 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'saved {saved} KB ({round(100 * saved / total_before, 1)}%)')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `before = {'js': 520, 'css': 110, 'image': 640}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of quantify before-and-after optimisation gains and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Summing each manifest and subtracting yields the bytes saved, and dividing by the original total expresses the improvement as a percentage.

```
before = {'js': 520, 'css': 110, 'image': 640}
after = {'js': 410, 'css': 90, 'image': 480}
total_before = sum(before.values())
```

```
    total_after = sum(after.values())
    saved = total_before - total_after
    print(f'before = {total_before} KB, after = {total_after} KB')
    print(f'saved {saved} KB ({round(100 * saved / total_before, 1)}%)')
```

```
output
before = 1270 KB, after = 980 KB
saved 290 KB (22.8%)
```

How it works:

1. before and after dicts share keys js, css, and image.
2. sum() totals each, giving 1270 and 980 KB, and the difference is 290 KB saved.
3. The print expresses the saving both as 290 KB and as 22.8 percent of the before total.

Try it:

1. Print the per-type saving so you can show which category improved most.
2. Guard against a zero before-total to avoid dividing by zero.
3. Flag any type that grew rather than shrank, which would be a regression hidden in the total.
4. Make the JS larger after the pass and watch the saving shrink.

⚠ Common Mistakes

- Dividing the saving by the after-total instead of the before-total overstates the gain.
- Reporting a negative saving as a win when a type actually grew.
- Forgetting to round the percentage gives a long, noisy decimal.

Skills gained before/after diff | savings percentage | summing values

→ Where You'll Use This: Quantifying a saving is how you report optimisation results to stakeholders, converting engineering effort into a percentage win that justifies the work.

Mental Model

Like stepping on the scale before and after a diet: the pounds lost and the percent of starting weight together make the result undeniable.

Project 298 Audit Core Web Vitals Across a Page Set

The Idea Behind It

A site passes Core Web Vitals when most of its pages pass individually. This project stores a metrics dict per page, runs the all-three-good test over each using all() against a shared thresholds dict, prints each page's verdict, then tallies the passers. Only /home clears all three, so the summary reads 1 of 3 pages passing.

What You'll Build & Learn

Run the all-three-good test over a set of pages and tally how many pass Core Web Vitals.

- Site pass depends on per-page passes
- all() applies the three-good test
- Tally the passing pages

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 297: quantify before-and-after optimisation gains. Project 299 changes the surface problem while preserving the underlying method.

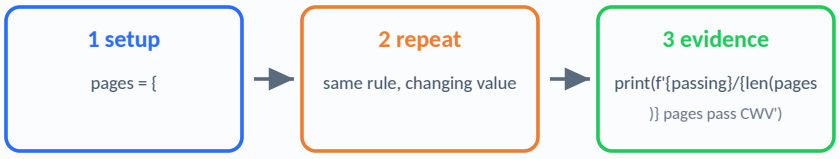
progress check

Reading target: audit core web vitals across a page set is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{passing}/{len(pages)} pages pass CWV')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`print(f'{path}: {"PASS" if passed else "FAIL"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{passing}/{len(pages)} pages pass CWV')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{path}: {"PASS" if passed else "FAIL"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Audit Core Web Vitals Across a Page Set: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: pages, home, lcp, inp, cls. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = { | Core: print(f'{path}: {"PASS" if passed else "FAIL"}') | Evidence: print(f'{passing}/{len(pages)} pages pass CWV')

🛠️ Reader to Writer Bridge

1. Read: explain audit core web vitals across a page set in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 297.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 299 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{passing}/{len(pages)} pages pass CWV')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of audit core web vitals across a page set and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Looping the pages, applying the all-metrics-good test, and counting the passers turns per-page verdicts into a site coverage figure.

```
pages = {
    '/home': {'lcp': 2.2, 'inp': 160, 'cls': 0.05},
    '/blog': {'lcp': 3.8, 'inp': 220, 'cls': 0.08},
    '/shop': {'lcp': 4.5, 'inp': 540, 'cls': 0.31},
}
th = {'lcp': 2.5, 'inp': 200, 'cls': 0.1}
for path in sorted(pages):
    m = pages[path]
    passed = all(m[k] <= th[k] for k in th)
    print(f'{path}: {"PASS" if passed else "FAIL"}')
passing = sum(1 for m in pages.values() if all(m[k] <= th[k] for k in th))
print(f'{passing}/{len(pages)} pages pass CWV')
```

```
output
/blog: FAIL
/home: PASS
```

```
/shop: FAIL
1/3 pages pass CWV
```

How it works:

1. A pages dict maps three paths each to a metrics dict, with a shared thresholds dict th.
2. For each sorted path, all(m[k] <= th[k] for k in th) decides PASS or FAIL and prints it.
3. A generator counts the passing pages, printing 1 of 3 pass CWV.

Try it:

1. Print the pass rate as a percentage rather than a fraction.
2. List only the failing pages and the specific metrics that broke them.
3. Group pages by template prefix and report a pass rate per template.
4. Fix /blog's LCP to 2.4 and watch the passing count rise.

⚠ Common Mistakes

- Recomputing the per-page pass twice (once to print, once to count) risks them disagreeing.
- Iterating pages unsorted yields nondeterministic per-page output order.
- Counting any good metric as a page pass instead of requiring all three.

Skills gained page-set audit | all() test | pass tallying

→ Where You'll Use This: Auditing a whole page set is how SEO teams report site-level Core Web Vitals health, spotting which templates drag the overall pass rate down.

Mental Model

Like grading a class on how many students passed the exam: a few star scores mean nothing if most of the room failed the same test.

Project 299 Generate Targeted Optimisation Hints

The Idea Behind It

An audit only earns its keep if it tells you what to do next. This project scans a list of resource dicts and emits targeted hints: compress when an asset is compressible and over 100 KB, cache when it is uncached and over 200 KB. Collecting and printing the hints turns raw resource flags into a concrete, prioritised action list.

What You'll Build & Learn

Scan resources for compressible-and-large or uncached-and-large cases and emit concrete fix hints.

- Audits should output actions

- Compound conditions trigger each hint
- Collect hints in a list to count them

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 298: audit core web vitals across a page set. Project 300 changes the surface problem while preserving the underlying method.

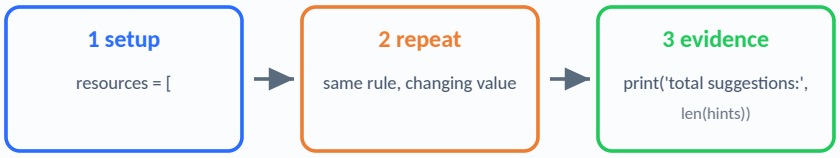
progress check

Reading target: generate targeted optimisation hints is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `resources = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('total suggestions:', len(hints))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`resources = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`if r['compressible'] and r['size'] > 100:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print('total suggestions:', len(hints))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if r['compressible'] and r['size'] > 100:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Generate Targeted Optimisation Hints: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: resources, url, app, js, size. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: resources = [| Core: if r['compressible'] and r['size'] > 100: | Evidence: print('total suggestions:', len(hints))

Reader to Writer Bridge

1. Read: explain generate targeted optimisation hints in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 298.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 300 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('total suggestions:', len(hints))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `resources = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of generate targeted optimisation hints and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Two guarded conditions per resource emit a compress or cache hint only when it would actually help, so the suggestion list stays actionable and short.

```
resources = [
    {'url': 'app.js', 'size': 300, 'compressible': True, 'cached': False},
    {'url': 'hero.jpg', 'size': 480, 'compressible': False, 'cached': False},
    {'url': 'style.css', 'size': 90, 'compressible': True, 'cached': True},
]
hints = []
for r in resources:
    if r['compressible'] and r['size'] > 100:
        hints.append(f"compress {r['url']}")
    if not r['cached'] and r['size'] > 200:
        hints.append(f"cache {r['url']}")
for h in hints:
    print('-', h)
print('total suggestions:', len(hints))
```

```
output
- compress app.js
- cache app.js
- cache hero.jpg
total suggestions: 3
```

How it works:

1. resources is a list of dicts with size, compressible, and cached flags.

2. Per resource, a compressible-and-over-100 test appends a compress hint and an uncached-and-over-200 test appends a cache hint.
3. It prints each hint with a dash and the total of three suggestions.

Try it:

1. Add a resize hint for images above a width or size limit.
2. Sort the hints so the highest-impact fixes appear first.
3. Attach the estimated saving to each hint so the list is prioritised by payoff.
4. Mark hero.jpg as cached and watch one hint disappear.

⚠ Common Mistakes

- Suggesting compression for an already-compressed image yields a useless hint.
- Emitting a cache hint for a tiny asset where the win is negligible.
- Counting hints by resources scanned rather than hints actually produced.

Skills gained hint generation | compound conditions | actionable audit

→ Where You'll Use This: Generating targeted hints is the bridge from measurement to remediation, the part of an audit tool that hands developers a ready to-do list rather than raw numbers.

Mental Model

Like a car inspection that does not just list faults but writes the work order: rotate tyres, change oil, ready for the mechanic to act on directly.

Project 300 Chart a Vitals Trend Across Weeks

The Idea Behind It

Tracking how many of the three vitals are good each week shows whether optimisation work is paying off. This project iterates a weeks dict, counts the good metrics per week by testing each against a shared thresholds dict with a generator, and prints the running tally. The climb from 0 of 3 to 3 of 3 makes the improving trajectory visible.

What You'll Build & Learn

Count how many of the three vitals are good per week to chart whether optimisation is working.

- Weekly good-counts show a trajectory
- Generator counts metrics under threshold
- Compare weeks at a glance

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 299: generate targeted optimisation hints. Project 301 changes the surface problem while preserving the underlying method.

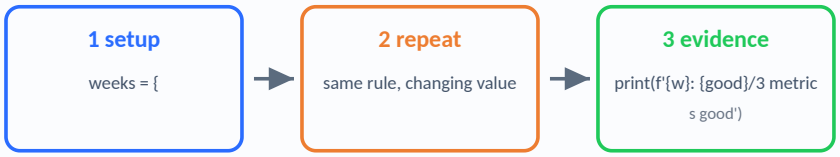
progress check

Reading target: chart a vitals trend across weeks is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `weeks = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print(f'{w}: {good}/3 metrics good')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`weeks = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`good = sum(1 for k in th if m[k] <= th[k])`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{w}: {good}/3 metrics good')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `good = sum(1 for k in th if m[k] <= th[k])`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Chart a Vitals Trend Across Weeks: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: weeks, w1, lcp, inp, cls. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: weeks = { | Core: good = sum(1 for k in th if m[k] <= th[k]) | Evidence: print(f'{w}: {good}/3 metrics good')

Reader to Writer Bridge

1. Read: explain chart a vitals trend across weeks in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 299.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 301 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{w}: {good}/3 metrics good')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `weeks = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of chart a vitals trend across weeks and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For each week, counting the metrics that meet their threshold yields a 0-to-3 score, and reading the weeks in order reveals the improvement trend.

```
weeks = {
    'w1': {'lcp': 4.2, 'inp': 380, 'cls': 0.22},
    'w2': {'lcp': 3.6, 'inp': 300, 'cls': 0.15},
    'w3': {'lcp': 2.4, 'inp': 190, 'cls': 0.08},
}
th = {'lcp': 2.5, 'inp': 200, 'cls': 0.1}
for w in sorted(weeks):
    m = weeks[w]
    good = sum(1 for k in th if m[k] <= th[k])
    print(f'{w}: {good}/3 metrics good')
```

```
output
w1: 0/3 metrics good
w2: 0/3 metrics good
w3: 3/3 metrics good
```

How it works:

1. A weeks dict maps w1, w2, and w3 to per-metric readings, with a shared thresholds dict th.
2. For each sorted week a generator counts the metrics at or under their threshold.
3. It prints the good count out of three per week, climbing from 0 to 3 by w3.

Try it:

1. Also name which specific metrics turned good each week.
2. Print an arrow showing whether the count improved versus the prior week.
3. Add a fourth week that regresses and confirm the count drops.
4. Add a week 4 that is all good and watch the score hit 3/3.

⚠ Common Mistakes

- Iterating the weeks dict unsorted scrambles the chronological trend.

- Counting metrics below threshold instead of at-or-below flips the score.
- Hard-coding 3 as the denominator instead of `len(th)` breaks if a metric is added.

Skills gained `weekly trend` `good-metric count` `generator sum`

→ **Where You'll Use This:** Charting good-metric counts over time is how teams demonstrate that performance work is moving the needle, ideal for a recurring stakeholder report.

⚡ Mental Model

Like tracking how many of three new habits you kept each week: watching the count rise from zero toward three is proof the routine is finally sticking.

Project 301 Run a Full Page Performance Audit

💡 The Idea Behind It

This capstone ties the chapter together by combining a request manifest with a metrics snapshot into one audit. It totals transfer against a budget, counts render-blocking resources, lists the failing vitals against their thresholds, and prints a single verdict. The page busts its budget and fails all three vitals, so the overall verdict reads NEEDS WORK.

What You'll Build & Learn

Combine a manifest and a metrics snapshot to total transfer, count blockers, list failing vitals, and decide a verdict.

- A full audit fuses bytes and vitals
- One verdict needs several checks to pass
- Manifest tuples carry size and a blocking flag

🧭 Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 300: chart a vitals trend across weeks. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run a full page performance audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

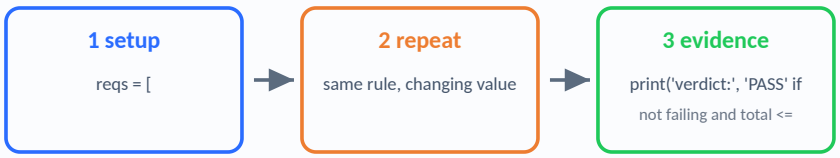
🔎 Logic Lens

- Setup: `reqs = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
- Evidence: `print('verdict:', 'PASS' if not failing and total <= budget_kb else 'NEEDS WORK')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`reqs = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
2. Apply the rule	`blocking = sum(1 for _, _, _, b in reqs if b)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('verdict:', 'PASS' if not failing and total <= budget_kb else 'NEEDS WORK')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Site Speed and Core Web Vitals becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `blocking = sum(1 for _, _, _, b in reqs if b)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run a Full Page Performance Audit: connect the surface task to the chapter skill, score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.
Names	Watch these names first: reqs, page, html, document, app. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: reqs = [| Core: blocking = sum(1 for _, _, _, b in reqs if b) | Evidence: print('verdict:', 'PASS' if not failing and total <= budget_kb else 'NE...

🛠️ Reader to Writer Bridge

1. Read: explain run a full page performance audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 300.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('verdict:', 'PASS' if not failing and total <= budget_kb else 'NEEDS WORK')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `reqs = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run a full page performance audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A single pass over the manifest yields the request count, transfer weight and blocking count, while the vitals check supplies the failing list; the verdict combines both gates.

```
reqs = [
    ('page.html', 'document', 20, False),
    ('app.js', 'script', 300, True),
    ('vendor.js', 'script', 180, False),
    ('style.css', 'stylesheet', 80, True),
    ('hero.jpg', 'image', 480, False),
    ('font.woff2', 'font', 90, True),
]
metrics = {'lcp': 3.4, 'inp': 210, 'cls': 0.12}
th = {'lcp': 2.5, 'inp': 200, 'cls': 0.1}
budget_kb = 1000
total = sum(s for _, _, s, _ in reqs)
blocking = sum(1 for _, _, _, b in reqs if b)
failing = [k for k in th if metrics[k] > th[k]]
print('=== Performance Audit ===')
print(f'fetches: {len(reqs)}, transfer: {total} KB / {budget_kb} KB')
print(f'render-blocking: {blocking}')
print(f'failing CWV: {sorted(failing) if failing else "none"}')
print('verdict:', 'PASS' if not failing and total <= budget_kb else 'NEEDS
WORK')
```

```
output
=== Performance Audit ===
fetches: 6, transfer: 1150 KB / 1000 KB
render-blocking: 3
failing CWV: ['cls', 'inp', 'lcp']
verdict: NEEDS WORK
```

How it works:

1. reqs is a list of tuples with url, type, size, and a blocking boolean, plus a metrics snapshot and thresholds.
2. It sums transfer to 1150 KB, counts three blocking resources, and lists every failing vital against th.
3. The verdict prints NEEDS WORK because the page is over budget and all three vitals fail.

Try it:

1. Add the per-type budget breakdown so the report names which category overshot.
2. Print the specific render-blocking URLs, not just the count.

3. Return a structured result dict instead of printing, so a caller can act on it programmatically.
4. Shrink hero.jpg so transfer drops under budget and re-check the verdict.

⚠ Common Mistakes

- Passing the audit while still over the transfer budget because only vitals were checked.
- Counting blocking resources but never feeding them into the final verdict.
- Reporting failing vitals unsorted so the audit output varies run to run.

Skills gained full audit | transfer total | verdict logic

→ Where You'll Use This: A combined audit like this is the heart of a real performance-monitoring script, the single function that ingests a page snapshot and returns a ship-or-fix decision.

Mental Model

Like a pre-flight checklist that touches fuel, weight, and instruments: every line must clear before the plane is cleared for takeoff, and one red item grounds it.

✅ Chapter Progress Checkpoint

What this chapter added: after 38 mini-projects, Site Speed and Core Web Vitals should feel like a reusable Python move: score lcp, inp and cls against thresholds, set performance budgets, and weigh page resources.

chapter checkpoint

1 Read
Open any project from Projects 264-301 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 9 (Structured Data and Rich Results) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

9 questions · 4 categories · graded

3 Recall | 2 Meaning | 2 Structure | 2 Transfer

8.1.Q1 Recall MEDIUM Turn back to Project 274 (Grade Time To First Byte). Without running it, what exactly does it print?

8.1.Q2 Meaning MEDIUM Turn back to Project 269 (Check Total Transfer Against a Budget). Without running it, what exactly does it print?

8.1.Q3 Structure MEDIUM Turn back to Project 267 (Decide the Overall Core Web Vitals Pass). Without running it, what exactly does it print?

8.1.Q4 Transfer MEDIUM Turn back to Project 290 (Compute Total Blocking Time From Tasks). Without running it, what exactly does it print?

8.1.Q5 Recall MEDIUM Which Core Web Vital measures how quickly a page visibly responds to a user interaction such as a tap or click?

A LCP

B INP

C CLS

D TTFB

8.1.Q6 Meaning MEDIUM A page reports LCP 2.4s (good), INP 150ms (good) and CLS 0.30 (poor). Does it pass Core Web Vitals?

A Yes — two of three are good

B Yes — the average is fine

C No — all three must be in the good band

D Only on desktop

8.1.Q7 Structure EASY Why does Google grade the field data at the 75th percentile (p75) rather than the average?

A Averages are harder to compute

B p75 ensures a slow tail of real visits cannot be hidden by a few fast loads

C p75 is always lower than the average

D It matches the Lighthouse algorithm

8.1.Q8 Transfer MEDIUM A page is 100 KB over its total transfer budget and three resources in the head are render-blocking. Which gives the most reliable picture of why LCP is poor?

8.1.Q9 Recall EASY State the good-band thresholds for all three Core Web Vitals.

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 8 – Site Speed and Core Web Vitals

8.1.Q1. `TTFB 640ms -> slow`

8.1.Q2. `total transfer = 1000 KB, budget = 900 KB -> OVER budget` then `headroom = -100 KB`

8.1.Q3. `Core Web Vitals: PASS`

8.1.Q4. `long tasks (> 50ms): 3` then `total blocking time = 250ms`

8.1.Q5. B Interaction to Next Paint (INP) measures responsiveness in milliseconds; it replaced FID in March 2024. LCP is load speed, CLS is visual stability.

8.1.Q6. C passing requires every vital in the good band at once, so the poor CLS fails the whole assessment.

8.1.Q7. B the average can be flattered by fast visits; the p75 reflects the experience of the slower three-quarters, the visits most likely to bounce.

8.1.Q8. Both the per-type weight breakdown (which category blew the budget) and the count of render-blocking resources (what delays first paint) — vitals are symptoms, while weight and blocking resources are the cause you can act on.

8.1.Q9. LCP ⩽ 2.5s, INP ⩽ 200ms, and CLS ⩽ 0.1. (The needs-improvement upper bounds are 4.0s, 500ms and 0.25 respectively.)

Chapter 9 · Structured Data and Rich Results

Generate and validate JSON-LD for Article, Product, FAQ, LocalBusiness and Breadcrumb.

Chapter Skill Map

What this chapter adds: In Structured Data and Rich Results, the focus is generate and validate json-ld for article, product, faq, localbusiness and breadcrumb. You practise it across 37 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples.

chapter arc

What it builds on: Chapter 9 assumes the reading habits from Chapter 8: Site Speed and Core Web Vitals. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 10, International SEO and hreflang, reuses this chapter's discipline with a new kind of problem.

1 **Projects 302-313**	2 **Projects 314-325**	3 **Projects 326-338**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Structured data is a fact sheet for robots: a JSON object that says *what* this page is.

@context names the vocabulary, @type names the thing

A crawler reads your prose the way a stranger skims a poster — it can guess that "$89.99" is probably a price and "Jane Doe" is probably a name, but it is guessing. Structured data removes the guessing by attaching a small, machine-readable fact sheet to the page: a labelled object that states *this is a Product, its name is Trail Runner Shoe, its price is 89.99 USD, and it is in stock.*

Today that fact sheet is almost always written as JSON-LD — JSON for Linked Data — a plain JSON object dropped into a `<script type="application/ld+json">` tag in the page. Because it is just JSON, you build it the same way you build any dictionary. In Python a Product fact sheet is a `dict`, and `json.dumps` is the entire publishing step:

```
import json
article = {
    "@context": "https://schema.org",
    "@type": "Article",
    "headline": "How Crawlers Work",
    "author": "Jane Doe",
    "datePublished": "2026-01-15",
}
print(json.dumps(article, indent=2))
```

```
output
{
  "@context": "https://schema.org",
  "@type": "Article",
  "headline": "How Crawlers Work",
  "author": "Jane Doe",
  "datePublished": "2026-01-15"
}
```

Two keys make this more than an ordinary dictionary. `@context` tells the engine which vocabulary the field names come from — for SEO that is always `"https://schema.org"`, the shared dictionary Google, Bing and others agree on. `@type` tells the engine *what kind of thing* the object describes, drawn from that vocabulary: `Article`, `Product`, `FAQPage`, and so on. Once the engine knows the context and the type, every other key (`headline`, `author`, `price`) has an agreed meaning.

without @type the engine has labels but no idea what they label

Each type publishes a small contract: some properties are `required` (omit one and the block is rejected) and some are `recommended` (omit one and you simply qualify for a plainer result). Satisfy the required set and your page becomes eligible for a `rich result` — the enhanced listing with stars, prices, FAQ accordions or breadcrumbs instead of a bare blue link. Structured data does not improve your ranking directly; it changes what your listing *looks like*, and a richer listing earns more clicks.

STRUCTURED DATA

A machine-readable description of a page's content, written in an agreed vocabulary so a search engine reads facts (price, author, rating) instead of inferring them from prose.

JSON-LD

JSON for Linked Data: the format Google recommends for structured data. A JSON object placed in a `<script type="application/ld+json">` tag, kept separate from the visible HTML.

@CONTEXT

The key that names the vocabulary the field names come from. For SEO it is always the string `"https://schema.org"`.

@TYPE

The key that names what the object is — `Article`, `Product`, `FAQPage`, `BreadcrumbList`, `LocalBusiness`, etc. An unknown or misspelled type is silently ignored.

SCHEMA.ORG

The shared vocabulary of types and properties maintained jointly by the major search engines. It defines, per type, which properties exist and which are required.

RICH RESULT

An enhanced search listing — stars, prices, FAQ accordion, breadcrumb trail — granted to a page whose structured data is valid and carries the required properties for its type.

Every block declares a context and a type

A JSON-LD block is checked for two keys before anything else. `@context` says which vocabulary the names belong to; `@type` says what the object is. Miss either and the engine cannot interpret the rest, so the very first audit is a presence check on those two keys:

```
data = {
    "@context": "https://schema.org",
    "@type": "Article",
    "headline": "How Crawlers Work",
}
has_ctx = "@context" in data
has_type = "@type" in data
ok = has_ctx and has_type
print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'invalid'}")
```

```
output
@context=True @type=True -> valid
```

The type also has to be a *real* type. A typo like `Artical` or `Prodcut` is not an error — the engine simply does not recognise it and throws the whole block away. Validating `@type` against a set of known types catches the silent failure:

```
KNOWN = {"Article", "Product", "FAQPage", "BreadcrumbList", "LocalBusiness"}
docs = [{"@type": "Article"}, {"@type": "Artical"}, {"@type": "Prodcut"}]
for d in docs:
    t = d["@type"]
    print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")
```

```
output
Article: known
Artical: unknown type
Prodcut: unknown type
```

Required vs recommended: the contract per type

Each schema type ships a contract. `Required` properties must be present or the block earns no rich result at all; `recommended` properties are optional but unlock a richer display. A data-driven validator keeps each type's required list in one table and checks any block by looking up its `@type` — so one function scales to dozens of types:

```
SCHEMAS = {
    "Article": ["headline", "author", "datePublished"],
    "Product": ["name", "offers"],
}
article = {"@context": "https://schema.org", "@type": "Article",
           "author": "Jane Doe"}
required = SCHEMAS[article["@type"]]
missing = [f for f in required if f not in article]
print("missing required:", missing)
```

```
output
missing required: ['headline', 'datePublished']
```

Recommended fields are audited the same way, but a miss is advice rather than a failure. Reporting the absent recommended fields tells an author exactly what to add next to upgrade from a basic to a richer result — an image, a description, a brand, a rating:

required = pass/fail · recommended = better/worse

```
product = {"@type": "Product", "name": "Widget",
           "offers": {"price": "5.00", "priceCurrency": "USD"}}
recommended = ["image", "description", "brand", "aggregateRating"]
missing = sorted(f for f in recommended if f not in product)
print("recommended missing:", missing)
```

```
output
recommended missing: ['aggregateRating', 'brand', 'description', 'image']
```

Nesting: properties whose values are themselves typed objects

Many properties do not take a plain string — they take another typed object. A Product's `offers` is an `Offer` with its own `price`, `priceCurrency` and `availability`; a LocalBusiness's `address` is a nested `PostalAddress`. The rule is simple: whenever a value has its own structure, it gets its own `@type` and becomes a sub-dictionary. Here a Product earns shopping eligibility by carrying a complete Offer:

```
import json
product = {
    "@context": "https://schema.org",
    "@type": "Product",
    "name": "Trail Runner Shoe",
    "offers": {
        "@type": "Offer",
```

```
        "price": "89.99",
        "priceCurrency": "USD",
        "availability": "https://schema.org/InStock",
    },
}
print(json.dumps(product["offers"], indent=2))
```

```
output
{
  "@type": "Offer",
  "price": "89.99",
  "priceCurrency": "USD",
  "availability": "https://schema.org/InStock"
}
```

The same nesting pattern builds the common list-shaped types. An `FAQPage` holds a `mainEntity` list where each entry is a `Question` whose `acceptedAnswer` is an `Answer`. A `BreadcrumbList` holds an `itemListElement` list of `ListItem` objects, each carrying a 1-based `position` so the engine can order the trail. A `LocalBusiness` feeds the map pack with a name, a nested address, a phone and opening hours. Build the breadcrumb from a plain Python list and the positions fall out of `enumerate`:

```
import json
trail = ["Home", "Catalog", "Shoes"]
crumbs = {
    "@context": "https://schema.org",
    "@type": "BreadcrumbList",
    "itemListElement": [
        {"@type": "ListItem", "position": i + 1, "name": name}
        for i, name in enumerate(trail)
    ],
}
for item in crumbs["itemListElement"]:
    print(item["position"], item["name"])
```

```
output
1 Home
2 Catalog
3 Shoes
```

Validating values, not just presence

A key being present is not enough — its *value* has to obey the schema's rules, or the rich result is dropped. Prices must be bare numbers (the currency lives in `priceCurrency`), currencies must be uppercase ISO 4217 codes, `availability` must be one of schema.org's fixed enum URLs, and dates must be ISO-8601 `YYYY-MM-DD`. These are the checks a validator runs after the presence pass. Availability is a good example — free text like `SoldOut` is rejected because it is not in the allowed enum:

```
allowed = {
    "https://schema.org/InStock",
    "https://schema.org/OutOfStock",
    "https://schema.org/PreOrder",
}
offers = [
    {"availability": "https://schema.org/InStock"},
    {"availability": "https://schema.org/SoldOut"},
]
for o in offers:
    a = o["availability"]
    print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enum")
```

```
output
InStock -> ok
SoldOut -> invalid enum
```

Pull it all together and a single audit decides `rich-result eligibility`: it confirms the context, the type, the required content fields, and at least one validated value, then reports an overall verdict. That combined check is exactly what tools like Google's Rich Results Test perform on a live page.

eligible only when every required check passes

```
product = {
    "@context": "https://schema.org",
    "@type": "Product",
    "name": "Trail Runner Shoe",
    "image": "https://example.com/shoe.jpg",
    "offers": {"@type": "Offer", "price": "89.99", "priceCurrency": "USD"},
}
checks = {
    "@context": product.get("@context") == "https://schema.org",
    "@type": product.get("@type") == "Product",
    "name": bool(product.get("name")),
    "image": bool(product.get("image")),
    "offer price": "price" in product.get("offers", {}),
}
for k, ok in checks.items():
    print(f"{k}: {'PASS' if ok else 'FAIL'}")
print("eligible:", all(checks.values()))
```

```
output
@context: PASS
@type: PASS
name: PASS
image: PASS
offer price: PASS
eligible: True
```

Example 1

You have the Q&A pairs ("What is JSON-LD?", "A way to embed structured data as JSON.") and ("Where does it go?", "In a script tag in the page head."). Build the first Question node of an FAQPage.

SOLUTION

1. An FAQPage carries a `mainEntity` list, and every entry is a `Question` object — so each pair becomes one node.
2. The question text goes in the Question's `name`; the answer is not a plain string but a nested `Answer` object under `acceptedAnswer`, holding the `text`.
3. So the first node is `{"@type": "Question", "name": "What is JSON-LD?", "acceptedAnswer": {"@type": "Answer", "text": "A way to embed structured data as JSON."}}`.

The Question wraps the answer in a typed Answer sub-object.

each Q&A pair → one Question whose acceptedAnswer is an Answer

Example 2

An Article block has @context, @type Article and an author, but no headline and no datePublished. The required set for Article is headline, author, datePublished. Is it rich-result eligible, and what must you add?

SOLUTION

1. Look up the required list for the type `Article`: `headline, author, datePublished`.
2. Diff it against the block — `author` is present; `headline` and `datePublished` are `absent`.
3. A type is eligible only when *every* required property is present, so a non-empty missing list means not eligible.

Not eligible — it is missing headline and datePublished; add both (a non-empty title and an ISO-8601 date) to qualify. *required is all-or-nothing*

Example 3

A BreadcrumbList has items at positions 1, 2 and 4. The trail should be a continuous 1-based sequence. Is the ordering valid, and what is wrong?

SOLUTION

1. A valid BreadcrumbList numbers its items 1, 2, 3, … with no skips or repeats.
2. The full expected range up to the highest position is {1, 2, 3, 4}; the present positions are {1, 2, 4}.
3. Subtract the present set from the expected set: {1,2,3,4} − {1,2,4} = `{3}`, so position 3 is missing.

Invalid — there is a gap at position 3; the third crumb was dropped, breaking the chain.

compare present positions to the full expected range to find the gap

⚡ Mental-Math Shortcut

Name it, then describe it, then prove it. Every block answers three questions in order: *which vocabulary?* (`@context` = schema.org), *what is this?* (`@type`), and *what is true of it?* (the properties). A block fails at the earliest unanswered question — a missing `@type` makes the properties meaningless, and a missing *required* property makes the whole block ineligible no matter how rich the rest is. Audit context and type first, required fields next, value formats last.

⚠ Common Mistakes

- Misspelling @type (Artical, Prodcut) — the engine does not error, it silently ignores the unknown type and the whole block is dropped.
- Putting a currency symbol in price ("$19.99") — price must be a bare number; the symbol belongs to priceCurrency, which must be an uppercase ISO 4217 code like USD.
- Using free text for an enum field such as availability ("SoldOut") instead of the exact schema.org URL (https://schema.org/OutOfStock), so the value is rejected.
- Marking up content with structured data that is not actually visible on the page — search engines require the JSON-LD to match what the user sees, or they treat it as spam.

This chapter contains 37 hands-on mini-projects (Projects 302–338). Type each one out, run it, and experiment.

Project 302 Build an Article JSON-LD Object

💡 The Idea Behind It

Structured data starts as an ordinary Python dictionary. Here you assemble an Article with the two magic keys @context and @type alongside plain content fields like headline, author, and datePublished. Feeding that dict to json.dumps with indent=2 turns it into the exact JSON-LD text a page embeds, which is what makes the page eligible for an article rich result.

What You'll Build & Learn

Build a minimal Article JSON-LD object and serialize it to readable JSON.

- A JSON-LD block is just a dict
- The @context and @type keys
- Serializing with json.dumps and indent

🧭 Skill Roadmap

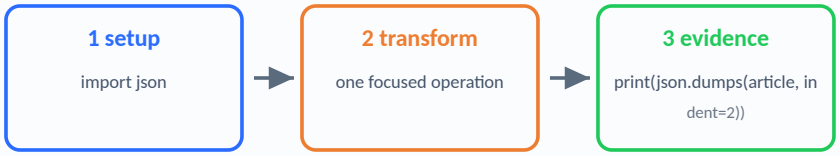
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 303 changes the surface problem while preserving the underlying method. *progress check*

Reading target: build an article json-ld object is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(json.dumps(article, indent=2))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(json.dumps(article, indent=2))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Build an Article JSON-LD Object: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, article, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print(json.dumps(article, indent=2))

🛠️ Reader to Writer Bridge

1. Read: explain build an article json-ld object in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 303 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(json.dumps(article, indent=2))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of build an article json-ld object and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A JSON-LD block is just a dict with `@context` pointing at schema.org and an `@type`; `json.dumps(..., indent=2)` renders the exact text a `<script type="application/ld+json">` tag would carry.

```
import json
article = {
    "@context": "https://schema.org",
    "@type": "Article",
    "headline": "How Crawlers Work",
    "author": "Jane Doe",
    "datePublished": "2026-01-15",
}
print(json.dumps(article, indent=2))
```

```
output
{
  "@context": "https://schema.org",
  "@type": "Article",
  "headline": "How Crawlers Work",
  "author": "Jane Doe",
  "datePublished": "2026-01-15"
}
```

How it works:

1. The article dict declares @context as the schema.org URL and @type as Article.
2. Three content fields follow: headline, author, and a datePublished string.
3. json.dumps with indent=2 prints the dict as nicely formatted JSON-LD text ready to paste into a page.

Try it:

1. Add an image field with an absolute URL and re-serialize the block
2. Swap @type to NewsArticle and add a publisher organization
3. Wrap the build in a function that takes headline and author as arguments
4. Add an `image` key and re-dump to see the block grow.

⚠ Common Mistakes

- Writing `@context` as `http://schema.org` trips network filters and looks unsafe; always use the `https://` form.
- Dumping without `indent` produces one long line that is valid but hard to eyeball in a review.

- By default `json.dumps` preserves insertion order, so the printed key order matches the dict literal, not alphabetical.

Skills gained `json.dumps` `dict literals` `JSON-LD` `schema.org`

→ Where You'll Use This: Every rich-result project starts from a dict like this; you generate article markup for blog posts and news pages so Google can show headline and byline.

⚡ Mental Model

The dict is the recipe and json.dumps is the printer that types the recipe onto the page in the format search engines read.

Project 303 Validate Context and Type Are Present

💡 The Idea Behind It

Before trusting any JSON-LD block you confirm it announces its vocabulary and its subject. This snippet uses the in operator to test whether @context and @type are present, stores the two booleans, ANDs them into a single ok flag, and prints a verdict. It mirrors the very first gate a structured-data validator applies to incoming markup.

What You'll Build & Learn

Check that a block declares both @context and @type before deeper validation.

- The in operator tests dict key membership
- Both @context and @type are mandatory
- Combining booleans with and

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 302: build an article json-ld object. Project 304 changes the surface problem while preserving the underlying method.

progress check

Reading target: validate context and type are present is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

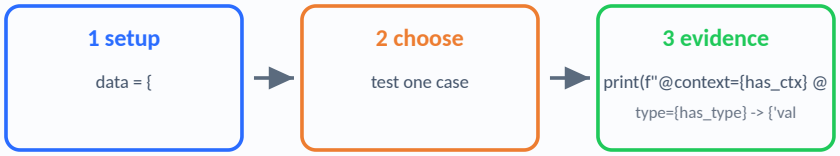
🔎 Logic Lens

- Setup: `data = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.

- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'invalid'}")` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`data = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'invalid'}")`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'invalid'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'invalid'}")`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Validate Context and Type Are Present: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: data, context, https, schema, org. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: data = { | Core: print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'inv... | Evidence: print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'inv...

🛠️ Reader to Writer Bridge

1. Read: explain validate context and type are present in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 302.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 304 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'invalid'}")`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `data = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate context and type are present and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Membership tests on the two mandatory keys are the cheapest possible structured-data check; a block missing either is ignored entirely by search engines.

```
data = {
    "@context": "https://schema.org",
    "@type": "Article",
    "headline": "How Crawlers Work",
}
has_ctx = "@context" in data
has_type = "@type" in data
ok = has_ctx and has_type
print(f"@context={has_ctx} @type={has_type} -> {'valid' if ok else 'invalid'}")
```

```
output
@context=True @type=True -> valid
```

How it works:

1. The data dict holds @context, @type, and a headline.
2. has_ctx and has_type each test key presence with the in operator.
3. ok ANDs the two flags, and an f-string with a conditional expression prints valid or invalid.

Try it:

1. Wrap the two checks in a function that returns the ok boolean
2. Loop over a list of blocks and count how many pass
3. Print which specific key is missing rather than a single verdict
4. Delete the `@type` key and confirm the verdict flips to invalid.

⚠ Common Mistakes

- Testing `data['@context']` instead of `'@context' in data` raises KeyError when the key is absent, which defeats the validation.
- Treating an empty-string `@type` as present passes the `in` check but is still invalid data.
- Checking only `@type` and forgetting `@context` lets a context-free block slip through.

Skills gained `in operator` `boolean logic` `f-strings` `key presence`

→ **Where You'll Use This:** This is the smoke test you run across a crawl of pages, flagging any JSON-LD missing its context or type before you bother checking the richer required fields.

⚡ Mental Model

Like checking a letter has both a language and a subject line before reading the body; without them the recipient cannot route it.

Project 304 Build a Product With an Offer

💡 The Idea Behind It

A Product earns a shopping rich result only when it carries pricing detail, and that detail lives in a nested Offer. This code builds a Product dict whose offers value is itself a dict with @type Offer, a price string, a priceCurrency, and a schema.org availability URL. Printing just the offers sub-dict shows the self-contained block that drives the price snippet.

What You'll Build & Learn

Build a Product whose nested Offer carries price, currency, and availability.

- Offers nest as a dict inside a Product
- Price, currency, and availability fields
- Availability uses a schema.org URL

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 303: validate context and type are present. Project 305 changes the surface problem while preserving the underlying method. *progress check*

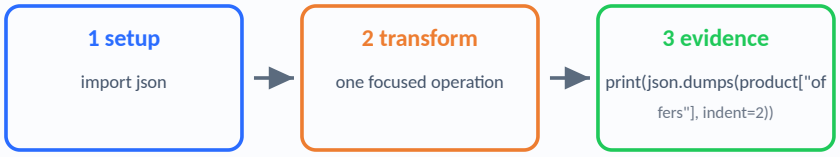
Reading target: build a product with an offer is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.

- Evidence: `print(json.dumps(product["offers"], indent=2))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(json.dumps(product["offers"], indent=2))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Build a Product With an Offer: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, product, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print(json.dumps(product["offers"], indent=2))

🛠️ Reader to Writer Bridge

1. Read: explain build a product with an offer in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 303.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 305 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(json.dumps(product["offers"], indent=2))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a product with an offer and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The `offers` value is itself a typed dict (`@type: Offer`) whose price, priceCurrency and availability are the trio Google needs for a price-rich result.

```
import json
product = {
    "@context": "https://schema.org",
    "@type": "Product",
    "name": "Trail Runner Shoe",
    "offers": {
        "@type": "Offer",
        "price": "89.99",
        "priceCurrency": "USD",
        "availability": "https://schema.org/InStock",
    },
}
print(json.dumps(product["offers"], indent=2))
```

```
output
{
  "@type": "Offer",
  "price": "89.99",
  "priceCurrency": "USD",
  "availability": "https://schema.org/InStock"
}
```

How it works:

1. The product dict has name and an offers key holding a nested Offer dict.
2. The Offer declares @type Offer, a price string, priceCurrency USD, and an InStock availability URL.
3. json.dumps prints only product['offers'], isolating the Offer block.

Try it:

1. Add a priceValidUntil date to the Offer and re-print
2. Make offers a list of two Offers for two sellers
3. Add a seller field pointing to a nested Organization
4. Add a `priceValidUntil` date to the offer.

⚠ Common Mistakes

- Encoding `price` as the float `89.99` instead of the string `"89.99"` is technically allowed but loses control over trailing-zero formatting.
- Writing `availability` as the bare word `InStock` instead of the full `https://schema.org/InStock` enum URL fails validation.
- Omitting `priceCurrency` makes the price meaningless and disqualifies the rich result.

Skills gained `nested dicts` `Offer schema` `json.dumps` `availability enum`

→ **Where You'll Use This:** On e-commerce product pages you generate exactly this Offer shape so the search result can show the price, currency symbol, and an in-stock badge beside the listing.

Mental Model

The Offer is the price tag clipped onto the product; the Product describes the item, the tag tells the shopper what it costs and whether it is on the shelf.

Project 305 Validate Required Product Fields

The Idea Behind It

Search engines only grant a Product rich result when its required fields are present. This snippet lists the required names, then uses a list comprehension to collect any that are not in the product dict. An empty missing list means the block passes; otherwise the names reveal what to add. It is required-field validation expressed in one comprehension.

What You'll Build & Learn

Detect which required Product fields are missing using a comprehension.

- A required-fields list drives validation
- List comprehension with an in filter
- Empty list means valid

Skill Roadmap

`Skill step:` adds one small variation while the chapter pattern is still new. It builds on Project 304: build a product with an offer. Project 306 changes the surface problem while preserving the underlying method.

progress check

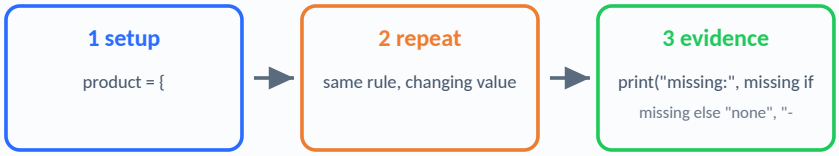
`Reading target:` validate required product fields is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- `Setup:` `product = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The comprehension compresses a loop into a readable noun phrase: `make this` for `each item` if `it qualifies`. Expand it mentally when it feels dense.
- `Writing move:` this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.

- Evidence: `print("missing:", missing if missing else "none", "->", "valid" if not missing else "invalid")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`product = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`missing = [f for f in required if f not in product]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print("missing:", missing if missing else "none", "->", "valid" if not missing else "invalid")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [f for f in required if f not in product]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate Required Product Fields: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: product, type, Product, name, Trail. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: product = { | Core: missing = [f for f in required if f not in product] | Evidence: print("missing:", missing if missing else "none", "->", "valid" if not...

🛠️ Reader to Writer Bridge

1. Read: explain validate required product fields in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 304.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 306 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("missing:", missing if missing else "none", "->", "valid" if not missing else "invalid")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `product = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate required product fields and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering the required-field list down to keys not present in the dict yields the missing set; an empty result means the block satisfies the schema's minimum.

```
product = {
    "@type": "Product",
    "name": "Trail Runner Shoe",
    "offers": {"price": "89.99", "priceCurrency": "USD"},
}
required = ["name", "offers"]
missing = [f for f in required if f not in product]
print("missing:", missing if missing else "none", "->", "valid" if not missing
else "invalid")
```

```
output
missing: none -> valid
```

How it works:

1. The product dict has name and a small offers dict.
2. required lists the field names name and offers.
3. The comprehension keeps each name not in product, and the print reports the missing names or none plus a valid/invalid verdict.

Try it:

1. Add price and priceCurrency to the required list and test against the nested offer
2. Return the missing list from a function instead of printing
3. Run the check across a list of three product dicts
4. Remove the `name` key and watch the verdict turn invalid.

⚠ Common Mistakes

- Checking `if not product[f]` instead of `if f not in product` raises KeyError on the very field you are testing for.
- Listing `price` as a top-level required field misses that it lives nested inside `offers`.
- Reporting the count of missing fields but not which ones leaves the author guessing.

Skills gained list comprehension | required fields | validation | membership test

→ Where You'll Use This: When auditing a catalog of product pages you run this to flag listings that will silently fail to earn a price snippet because a mandatory field never made it into the markup.

Mental Model

Like a checklist at a checkpoint: every box must be ticked or the product is turned away before it can appear with rich detail.

Project 306 Build an FAQPage From Q&A Pairs

The Idea Behind It

An FAQPage converts question and answer pairs into structured data so Google can show the answers as expandable entries. This code holds the pairs as tuples, then a list comprehension builds the mainEntity array, turning each tuple into a Question node whose acceptedAnswer is an Answer dict carrying the text. Printing the first node shows the nested Question/Answer shape.

What You'll Build & Learn

Transform a list of Q&A tuples into FAQPage mainEntity nodes.

- FAQPage holds a mainEntity list
- Each pair becomes a Question with an acceptedAnswer
- Comprehension over (q, a) tuples

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 305: validate required product fields. Project 307 changes the surface problem while preserving the underlying method.

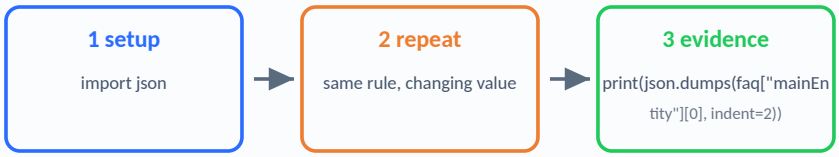
progress check

Reading target: build an faqpage from q&a pairs is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(json.dumps(faq["mainEntity"][0], indent=2))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`for q, a in qa`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(json.dumps(faq["mainEntity"][0], indent=2))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for q, a in qa`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build an FAQPage From Q&A Pairs: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, qa, What, is, LD. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: for q, a in qa | Evidence: print(json.dumps(faq["mainEntity"][0], indent=2))

🛠️ Reader to Writer Bridge

1. Read: explain build an faqpage from q&a pairs in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 305.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 307 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(json.dumps(faq["mainEntity"][0], indent=2))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build an faqpage from q&a pairs and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A list comprehension maps each `(question, answer)` tuple to a `Question` dict with a nested `Answer`, building the `mainEntity` array that an FAQPage requires.

```
import json
qa = [
    ("What is JSON-LD?", "A way to embed structured data as JSON."),
    ("Where does it go?", "In a script tag in the page head."),
]
faq = {
    "@context": "https://schema.org",
    "@type": "FAQPage",
```

```
    "mainEntity": [
        {"@type": "Question", "name": q,
         "acceptedAnswer": {"@type": "Answer", "text": a}}
        for q, a in qa
    ],
}
print(json.dumps(faq["mainEntity"][0], indent=2))
```

```
output
{
  "@type": "Question",
  "name": "What is JSON-LD?",
  "acceptedAnswer": {
    "@type": "Answer",
    "text": "A way to embed structured data as JSON."
  }
}
```

How it works:

1. qa is a list of (question, answer) tuples.
2. The faq dict declares FAQPage and builds mainEntity with a comprehension unpacking q, a from each tuple.
3. Each item is a Question node whose acceptedAnswer is an Answer dict holding the text; the first node is printed.

Try it:

1. Add a third Q&A tuple and confirm mainEntity grows
2. Read the pairs from a CSV instead of hard-coded tuples
3. Strip whitespace from each question and answer before building the node
4. Add a third tuple and confirm `mainEntity` grows.

⚠ Common Mistakes

- Putting the answer text directly on the Question instead of inside `acceptedAnswer` produces an FAQ that renders no answer.
- Naming the array `mainEntities` (plural) instead of `mainEntity` breaks the schema key.
- Reusing the same Question name for two pairs makes duplicate, confusing FAQ entries.

Skills gained tuple unpacking | list comprehension | FAQPage | nested nodes

→ Where You'll Use This: Support and help-center pages feed their FAQ content into this shape so the questions surface directly in search, capturing space and clicks for common queries.

Mental Model

The tuples are raw interview notes; the comprehension formats each exchange into a tidy question-and-answer card the engine can display.

Project 307 Count Question and Answer Pairs

The Idea Behind It

An FAQPage with a blank answer silently loses that entry, so an audit counts both questions and answers. This snippet pulls the mainEntity list, reports its length as the question count, then uses a generator inside sum with .get to count how many acceptedAnswer dicts actually carry text. Matching counts mean every question is answered.

What You'll Build & Learn

Count questions and verify each one carries a non-empty answer.

- len gives the question count
- sum over a generator counts matches
- dict.get guards a missing key

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 306: build an faqpage from q&a pairs. Project 308 changes the surface problem while preserving the underlying method.

progress check

Reading target: count question and answer pairs is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

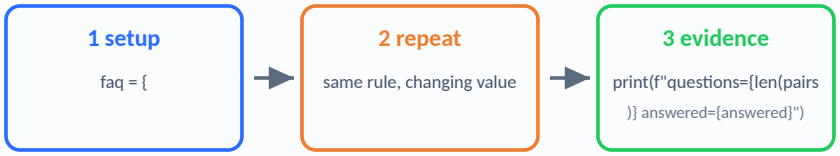
- Setup: `faq = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"questions={len(pairs)} answered={answered}")` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`faq = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`answered = sum(1 for q in pairs if q["acceptedAnswer"].get("text"))`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"questions={len(pairs)} answered={answered}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `answered = sum(1 for q in pairs if q["acceptedAnswer"].get("text"))`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Question and Answer Pairs: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: faq, type, FAQPage, mainEntity, Question. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: faq = { | Core: answered = sum(1 for q in pairs if q["acceptedAnswer"].get("text")) | Evidence: print(f"questions={len(pairs)} answered={answered}")

Reader to Writer Bridge

1. Read: explain count question and answer pairs in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 306.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 308 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"questions={len(pairs)} answered={answered}")`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `faq = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count question and answer pairs and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Counting `mainEntity` entries gives the question total, while summing those whose `acceptedAnswer.text` is truthy confirms none was left blank.

```
faq = {
    "@type": "FAQPage",
    "mainEntity": [
        {"@type": "Question", "name": "Q1", "acceptedAnswer": {"text": "A1"}},
        {"@type": "Question", "name": "Q2", "acceptedAnswer": {"text": "A2"}},
```

```
        {"@type": "Question", "name": "Q3", "acceptedAnswer": {"text": "A3"}},
    ],
}
pairs = faq["mainEntity"]
answered = sum(1 for q in pairs if q["acceptedAnswer"].get("text"))
print(f"questions={len(pairs)} answered={answered}")
```

```
output
questions=3 answered=3
```

How it works:

1. The faq dict's mainEntity holds three Question nodes, each with an acceptedAnswer.
2. pairs aliases that list and len(pairs) is the question count.
3. sum with a generator counts questions whose acceptedAnswer.get('text') is truthy, and the f-string prints both numbers.

Try it:

1. Print the name of any question whose answer is empty
2. Treat a whitespace-only answer as missing by stripping first
3. Fail loudly with an assert when answered is less than the question count
4. Blank out Q2's answer text and watch `answered` drop to 2.

⚠ Common Mistakes

- Using `q['acceptedAnswer']['text']` instead of `.get('text')` raises KeyError on a question that has no answer node.
- Counting a question with an empty-string answer as answered, since `''` is falsy but `'text' in dict` would be true.
- Iterating the FAQPage dict itself rather than its `mainEntity` list counts top-level keys.

Skills gained len | generator sum | dict.get | truthiness

→ Where You'll Use This: Before publishing an FAQ block you run this to catch a copy-paste slip where a question shipped with an empty answer, which would drop it from the rich result.

Mental Model

Like counting chairs and counting people seated: a mismatch means someone is standing, here a question with no answer attached.

Project 308 Build a BreadcrumbList With Positions

The Idea Behind It

A BreadcrumbList shows the path from the homepage down to the current page in the search snippet. This code takes a trail of names and builds itemListElement with a comprehension that enumerates them, giving each a ListItem with a 1-based position via i + 1. Looping the result prints each position next to its name, the ordering the engine relies on.

What You'll Build & Learn

Build a BreadcrumbList with correctly numbered ListItem positions.

- Breadcrumbs are an itemListElement list
- enumerate plus i + 1 gives 1-based positions
- Each crumb is a ListItem

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 307: count question and answer pairs. Project 309 changes the surface problem while preserving the underlying method.

progress check

Reading target: build a breadcrumblist with positions is the surface task; the transferable Python move is to mark setup, rule, and output before running.

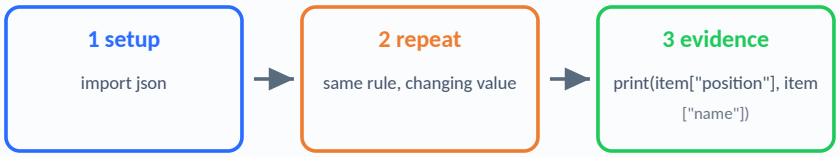
setup -> rule -> result

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(item["position"], item["name"])` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`for i, name in enumerate(trail)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(item["position"], item["name"])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for i, name in enumerate(trail)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a BreadcrumbList With Positions: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, trail, Home, Catalog, Shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import json | Core: for i, name in enumerate(trail) | Evidence: print(item["position"], item["name"])

Reader to Writer Bridge

1. Read: explain build a breadcrumblist with positions in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 307.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 309 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(item["position"], item["name"])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a breadcrumblist with positions and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

From Concept to Code (Python)

`enumerate` over the trail yields a zero-based index that becomes the 1-based `position` each ListItem needs, building an ordered breadcrumb array.

```
import json
trail = ["Home", "Catalog", "Shoes"]
crumbs = {
    "@context": "https://schema.org",
    "@type": "BreadcrumbList",
    "itemListElement": [
        {"@type": "ListItem", "position": i + 1, "name": name}
```

```
            for i, name in enumerate(trail)
        ],
    }
    for item in crumbs["itemListElement"]:
        print(item["position"], item["name"])
```

```
output
1 Home
2 Catalog
3 Shoes
```

How it works:

1. trail is the list of page names from Home to Shoes.
2. The comprehension uses enumerate so each name gets a ListItem with position i + 1 and the name.
3. A for loop prints each item's position and name, showing 1 Home, 2 Catalog, 3 Shoes.

Try it:

1. Add an item URL to each ListItem alongside name and position
2. Build the trail from a URL path split on slashes
3. Reverse the trail and confirm positions still start at one
4. Pass `enumerate(trail, 1)` and drop the `+ 1` for the same result.

⚠ Common Mistakes

- Using the raw `enumerate` index (starting at 0) gives a first position of 0, but schema.org positions start at 1.
- Passing `start=1` to `enumerate` and then also adding 1 double-counts the offset.
- Naming the array `itemListElements` (plural) breaks the required key.

Skills gained enumerate | list comprehension | BreadcrumbList | 1-based indexing

→ Where You'll Use This: Category and product pages generate breadcrumb markup so search results display the site hierarchy, helping users see where a page sits and improving click context.

Mental Model

Like numbering the steps of a staircase from the front door up; the engine needs the count so it can lay the crumbs in the right order.

Project 309 Validate Breadcrumb Position Ordering

The Idea Behind It

Breadcrumb positions must run 1, 2, 3 without skips or repeats or the engine may reject the trail. This snippet extracts the positions into a list, builds the expected sequence with range, and compares the two lists for exact equality. A True result confirms the numbering is clean; any deviation prints broken instead of ordered.

What You'll Build & Learn

Confirm breadcrumb positions form an unbroken 1-based sequence.

- Pull positions into a list
- range builds the expected sequence
- List equality compares order and values

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 308: build a breadcrumblist with positions. Project 310 changes the surface problem while preserving the underlying method. *progress check*

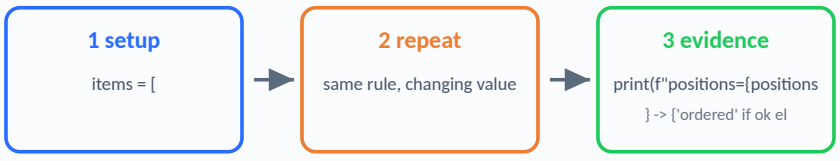
Reading target: validate breadcrumb position ordering is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `items = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"positions={positions} -> {'ordered' if ok else 'broken'}")` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`items = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"positions={positions} -> {'ordered' if ok else 'broken'}")`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"positions={positions} -> {'ordered' if ok else 'broken'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"positions={positions} -> {'ordered' if ok else 'broken'}")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate Breadcrumb Position Ordering: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: items, position, name, Home, Catalog. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: items = [| Core: print(f"positions={positions} -> {'ordered' if ok else 'broken'}") | Evidence: print(f"positions={positions} -> {'ordered' if ok else 'broken'}")

🛠️ Reader to Writer Bridge

1. Read: explain validate breadcrumb position ordering in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 308.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 310 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"positions={positions} -> {'ordered' if ok else 'broken'}")`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `items = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate breadcrumb position ordering and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Building the expected list `[1..n]` and comparing it for equality to the extracted positions verifies the breadcrumb is both complete and in order in one step.

```
items = [
    {"position": 1, "name": "Home"},
    {"position": 2, "name": "Catalog"},
    {"position": 3, "name": "Shoes"},
]
positions = [it["position"] for it in items]
expected = list(range(1, len(items) + 1))
ok = positions == expected
print(f"positions={positions} -> {'ordered' if ok else 'broken'}")
```

```
output
positions=[1, 2, 3] -> ordered
```

How it works:

1. items holds three crumbs with positions 1, 2, 3.
2. positions is a comprehension collecting each it['position'].

3. expected is list(range(1, len+1)), and comparing positions == expected yields the ordered or broken verdict.

Try it:

1. Sort the items by position before comparing in case they arrive out of order
2. Report the first index where actual and expected diverge
3. Test against a list whose positions are 1, 3, 2 and observe broken
4. Swap two items and watch the verdict turn broken.

⚠ Common Mistakes

- Comparing `sorted(positions)` to `expected` would call a shuffled `[3, 1, 2]` valid because sorting hides the disorder.
- Using `range(len(items))` (0-based) makes the expected list start at 0 and never match.
- Checking only the maximum position misses a missing middle item.

Skills gained range | list comparison | list comprehension | validation

→ Where You'll Use This: After generating breadcrumb markup at scale you verify each list is sequentially numbered, since a templating bug that repeats or skips a position quietly disqualifies the breadcrumb display.

⚡ Mental Model

Like checking a deck of numbered tickets is in order before the show; one missing or duplicate number and the sequence is invalid.

Project 310 Build a LocalBusiness Entity

💡 The Idea Behind It

A LocalBusiness block supplies the map pack and knowledge panel with the essentials of a storefront. This code assembles a dict with name, telephone, and an openingHours string, plus an address that is itself a nested PostalAddress object holding street, locality, and postal code. Serializing with json.dumps shows the full nested structure a local listing needs.

What You'll Build & Learn

Build a LocalBusiness with a nested PostalAddress, phone, and hours.

- LocalBusiness feeds local search features
- Address nests as a PostalAddress object
- openingHours uses a compact day-time string

Skill Roadmap

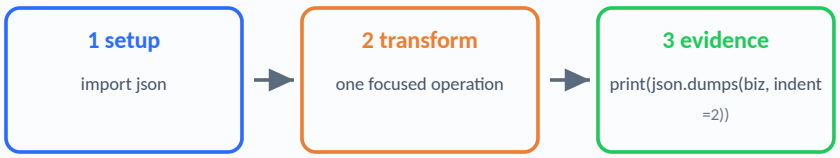
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 309: validate breadcrumb position ordering. Project 311 changes the surface problem while preserving the underlying method. *progress check*

Reading target: build a localbusiness entity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(json.dumps(biz, indent=2))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(json.dumps(biz, indent=2))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Build a LocalBusiness Entity: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, biz, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print(json.dumps(biz, indent=2))

🛠️ Reader to Writer Bridge

1. Read: explain build a localbusiness entity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 309.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 311 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(json.dumps(biz, indent=2))`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a localbusiness entity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The `address` is a typed `PostalAddress` sub-dict, and `openingHours` uses schema.org's compact `Mo-Fr 08:00-18:00` day/time notation that map listings parse directly.

```
import json
biz = {
    "@context": "https://schema.org",
    "@type": "LocalBusiness",
    "name": "Corner Cafe",
    "address": {
        "@type": "PostalAddress",
        "streetAddress": "12 Market St",
        "addressLocality": "Springfield",
        "postalCode": "10001",
    },
    "telephone": "+1-555-0100",
    "openingHours": "Mo-Fr 08:00-18:00",
}
print(json.dumps(biz, indent=2))
```

```
output
{
  "@context": "https://schema.org",
  "@type": "LocalBusiness",
  "name": "Corner Cafe",
  "address": {
    "@type": "PostalAddress",
    "streetAddress": "12 Market St",
    "addressLocality": "Springfield",
    "postalCode": "10001"
  },
  "telephone": "+1-555-0100",
  "openingHours": "Mo-Fr 08:00-18:00"
}
```

How it works:

1. The biz dict declares LocalBusiness with a name.
2. Its address key holds a nested PostalAddress with streetAddress, addressLocality, and postalCode.
3. telephone and openingHours follow, then json.dumps prints the whole nested block.

Try it:

1. Add geo coordinates as a nested GeoCoordinates object
2. Make openingHours a list covering weekend hours separately
3. Add a priceRange field and re-serialize the block
4. Add a `geo` object with latitude and longitude.

⚠ Common Mistakes

- Writing the address as one flat string instead of a `PostalAddress` object loses the structured locality and postcode.
- Using a human phrase like `Mon to Fri 8am` instead of the `Mo-Fr 08:00-18:00` format means opening hours will not parse.
- Storing `telephone` without a country code (`+1`) weakens the click-to-call match.

Skills gained nested objects | PostalAddress | LocalBusiness | json.dumps

→ Where You'll Use This: Multi-location businesses generate one of these per branch so each storefront can appear in the map pack with its address, phone number, and hours visible.

⚡ Mental Model

Like the storefront window display: name, address, phone, and hours all visible at a glance so a passerby knows where you are and when you open.

Project 311 Build an Organization With sameAs Links

💡 The Idea Behind It

An Organization block links a brand to its verified presence elsewhere through the sameAs array, helping engines assemble a knowledge panel. This code builds the org dict with name and url, then a sameAs list of canonical profile URLs. It prints the array and reports its length with len, the count of authoritative profiles the brand claims.

What You'll Build & Learn

Build an Organization that links to social profiles via sameAs.

- sameAs is an array of profile URLs
- It supports knowledge-panel assembly
- len counts the linked profiles

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 310: build a localbusiness entity. Project 312 changes the surface problem while preserving the underlying method.

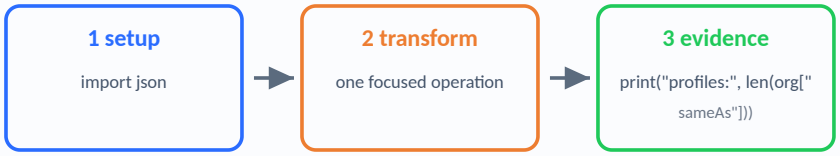
progress check

Reading target: build an organization with sameas links is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("profiles:", len(org["sameAs"]))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print("profiles:", len(org["sameAs"]))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Build an Organization With sameAs Links: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, org, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print("profiles:", len(org["sameAs"]))

Reader to Writer Bridge

1. Read: explain build an organization with sameas links in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 310.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 312 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("profiles:", len(org["sameAs"]))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build an organization with sameas links and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The `sameAs` array is a list of authoritative URLs for the same entity; counting its length tells you how many external profiles reinforce the brand's identity.

```
import json
org = {
    "@context": "https://schema.org",
    "@type": "Organization",
    "name": "Acme Inc",
    "url": "https://acme.example",
    "sameAs": [
        "https://twitter.com/acme",
        "https://www.linkedin.com/company/acme",
    ],
}
print(json.dumps(org["sameAs"], indent=2))
print("profiles:", len(org["sameAs"]))
```

```
output
[
  "https://twitter.com/acme",
  "https://www.linkedin.com/company/acme"
]
profiles: 2
```

How it works:

1. The org dict declares Organization with name and url.
2. sameAs is a list of two canonical profile URLs for Twitter and LinkedIn.
3. json.dumps prints the sameAs array and len reports profiles: 2.

Try it:

1. Add a logo field pointing to an absolute image URL
2. Append a Wikipedia URL to the sameAs list
3. Validate each sameAs entry starts with https before counting

4. Append a Facebook profile URL and watch the count rise.

⚠ Common Mistakes

- Putting a single profile as a bare string rather than a one-element list makes `len()` count characters, not profiles.
- Listing a non-canonical or redirecting profile URL weakens the entity link.
- Confusing `url` (the org's own site) with `sameAs` (external profiles) muddles the two fields.

Skills gained sameAs | Organization | list length | json.dumps

→ Where You'll Use This: Brand and about pages carry an Organization block whose sameAs entries point to official social and Wikipedia profiles, reinforcing entity identity for the knowledge graph.

⚡ Mental Model

sameAs is the brand saying these other accounts are also me; the engine connects the dots into one entity panel.

Project 312 Build a Recipe With Ingredients and Steps

💡 The Idea Behind It

A Recipe rich result wants three things together: an ingredient list, ordered steps, and a cook time in ISO-8601 duration form. This code builds the recipe dict with recipeIngredient as a list of strings and recipeInstructions as HowToStep objects so each step renders alone. It then prints the ingredient count, step count, and the cookTime value PT15M.

What You'll Build & Learn

Build a Recipe with ingredients, HowToStep instructions, and a cook time.

- recipeIngredient is a list of strings
- Steps are individual HowToStep objects
- cookTime uses ISO-8601 duration

🧭 Skill Roadmap

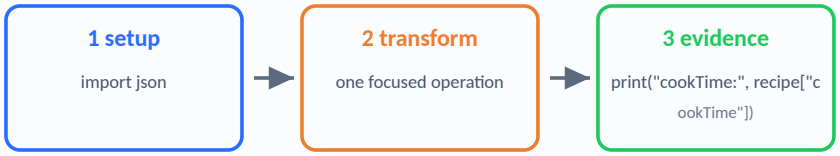
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 311: build an organization with sameas links. Project 313 changes the surface problem while preserving the underlying method. *progress check*

Reading target: build a recipe with ingredients and steps is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("cookTime:", recipe["cookTime"])` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print("cookTime:", recipe["cookTime"])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Build a Recipe With Ingredients and Steps: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, recipe, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print("cookTime:", recipe["cookTime"])

🛠️ Reader to Writer Bridge

1. Read: explain build a recipe with ingredients and steps in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 311.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 313 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("cookTime:", recipe["cookTime"])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of build a recipe with ingredients and steps and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`recipeIngredient` is a flat string list while `recipeInstructions` is a list of typed `HowToStep` objects, and `cookTime` uses the ISO duration `PT15M` (15 minutes).

```
import json
recipe = {
    "@context": "https://schema.org",
    "@type": "Recipe",
    "name": "Pancakes",
    "recipeIngredient": ["2 cups flour", "1 egg", "1 cup milk"],
    "recipeInstructions": [
        {"@type": "HowToStep", "text": "Mix the batter."},
        {"@type": "HowToStep", "text": "Cook on a griddle."},
    ],
    "cookTime": "PT15M",
}
print("ingredients:", len(recipe["recipeIngredient"]))
print("steps:", len(recipe["recipeInstructions"]))
print("cookTime:", recipe["cookTime"])
```

```
output
ingredients: 3
steps: 2
cookTime: PT15M
```

How it works:

1. The recipe dict declares Recipe with a name.
2. recipeIngredient is a three-item string list and recipeInstructions is a list of two HowToStep dicts.
3. len reports the ingredient and step counts, and cookTime PT15M is printed directly.

Try it:

1. Add a recipeYield field and print it
2. Add a nutrition object with calories
3. Convert each instruction to include a step name as well as text
4. Add a `prepTime` of `PT5M`.

⚠ Common Mistakes

- Writing `cookTime` as `15 minutes` instead of the ISO `PT15M` duration fails the rich-result test.

- Using the singular `recipeIngredient` for the list is correct schema.org naming; pluralizing it to `recipeIngredients` breaks it.
- Storing steps as plain strings instead of `HowToStep` objects loses per-step structure.

Skills gained HowToStep | Recipe schema | len | ISO-8601 duration

→ Where You'll Use This: Food and cooking sites markup each recipe this way so search can show the dish with its prep steps and total time, often with an image carousel.

Mental Model

The recipe card with three sections: a shopping list, a numbered method, and a clock for how long it takes.

Project 313 Build an Event With Date and Location

The Idea Behind It

An Event block puts dates and venues into search and maps. This code builds the event dict with a name and a startDate in ISO date-time form, then pairs it with a nested Place object describing the venue by name and address. Serializing with json.dumps reveals the typed nesting: an Event whose location is a Place, the shape an event listing needs.

What You'll Build & Learn

Build an Event with an ISO startDate and a nested Place location.

- Event surfaces date and venue
- startDate uses ISO date-time
- location nests as a Place object

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 312: build a recipe with ingredients and steps. Project 314 changes the surface problem while preserving the underlying method.

progress check

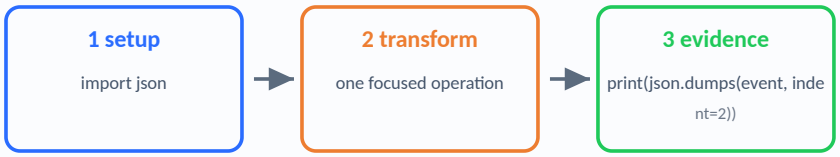
Reading target: build an event with date and location is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.

- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(json.dumps(event, indent=2))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(json.dumps(event, indent=2))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Build an Event With Date and Location: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, event, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print(json.dumps(event, indent=2))

🛠️ Reader to Writer Bridge

1. Read: explain build an event with date and location in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 312.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 314 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(json.dumps(event, indent=2))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build an event with date and location and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

The `startDate` is an ISO-8601 datetime string and `location` is a typed `Place`, the minimum pair that lets an Event show its when and where.

```
import json
event = {
    "@context": "https://schema.org",
    "@type": "Event",
    "name": "SEO Summit",
    "startDate": "2026-09-01T09:00",
    "location": {
        "@type": "Place",
        "name": "Convention Center",
        "address": "500 Expo Blvd",
    },
}
print(json.dumps(event, indent=2))
```

```
output
{
  "@context": "https://schema.org",
  "@type": "Event",
  "name": "SEO Summit",
  "startDate": "2026-09-01T09:00",
  "location": {
    "@type": "Place",
    "name": "Convention Center",
    "address": "500 Expo Blvd"
  }
}
```

How it works:

1. The event dict declares Event with a name.
2. startDate is an ISO date-time string and location is a nested Place with its own name and address.
3. json.dumps prints the full block showing the Place nested inside the Event.

Try it:

1. Add an endDate and an eventStatus field
2. Make location a VirtualLocation with a url for an online event
3. Add an offers block so tickets show a price
4. Add an `endDate` to make it a two-day event.

⚠ Common Mistakes

- Omitting the time and writing `startDate` as just `2026-09-01` is allowed but loses the start hour shown in results.
- Using a bare address string for `location` instead of a `Place` object skips the venue name.

- Forgetting `endDate` on a multi-day event makes it look like a single instant.

Skills gained Event schema | Place | ISO date-time | nested objects

→ Where You'll Use This: Concert, conference, and webinar pages emit Event markup so listings show the date and venue, and qualify for the dedicated events experience in search.

Mental Model

Like a calendar invite: a when (startDate) and a where (the nested Place), the two facts a listing must answer.

Project 314 Serialize JSON-LD as a Pretty String

The Idea Behind It

Auditors want a stable rendering of a block so diffs stay meaningful run to run. This snippet builds a WebSite dict and calls json.dumps with both indent=2 and sort_keys=True. Sorting keys plus fixed indentation yields canonical text: the same dict always produces byte-identical output, which makes change detection across crawls reliable.

What You'll Build & Learn

Serialize a block to canonical, diff-friendly JSON with sorted keys.

- sort_keys makes output order deterministic
- indent makes it readable
- Canonical text enables stable diffs

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 313: build an event with date and location. Project 315 changes the surface problem while preserving the underlying method.

progress check

Reading target: serialize json-ld as a pretty string is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

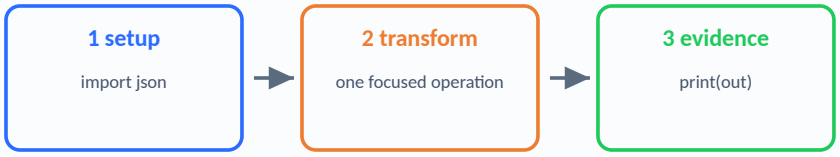
Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.

- Evidence: `print(out)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(out)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Serialize JSON-LD as a Pretty String: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, data, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print(out)

🛠️ Reader to Writer Bridge

1. Read: explain serialize json-ld as a pretty string in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 313.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 315 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(out)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of serialize json-ld as a pretty string and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`sort_keys=True` orders keys alphabetically (so `@` symbols sort before letters), giving a canonical form ideal for diffing two versions of a block.

```
import json
data = {
    "@context": "https://schema.org",
    "@type": "WebSite",
    "name": "My Blog",
    "url": "https://blog.example",
```

```
}
out = json.dumps(data, indent=2, sort_keys=True)
print(out)
```

```
output
{
  "@context": "https://schema.org",
  "@type": "WebSite",
  "name": "My Blog",
  "url": "https://blog.example"
}
```

How it works:

1. The data dict declares WebSite with name and url.
2. json.dumps is called with indent=2 and sort_keys=True.
3. The result is printed; key order is alphabetical regardless of insertion order, so identical data prints identically every time.

Try it:

1. Compare output with and without sort_keys on the same dict
2. Hash the canonical string with hashlib to fingerprint the block
3. Add nested keys and confirm inner dicts also sort
4. Drop `sort_keys` and compare the key order.

⚠ Common Mistakes

- Leaving out `sort_keys` makes the output depend on insertion order, which breaks byte-for-byte diffs.
- Setting `indent=0` still adds newlines; only `indent=None` produces a single compact line.
- Assuming `@context` sorts after `name`, when in fact `@` (0x40) sorts before lowercase letters.

Skills gained json.dumps | sort_keys | canonical output | serialization

→ Where You'll Use This: When you snapshot a site's structured data over time, canonical serialization lets you diff today's markup against yesterday's and spot exactly what a deploy changed.

Mental Model

Like alphabetizing a filing cabinet: once everything has a fixed place, you can instantly see if a single file moved.

Project 315 Detect a Missing Required Property

The Idea Behind It

Each schema type carries a small set of required properties, and this snippet looks them up by type. The required table maps Article to its three mandatory fields; the comprehension reads the block's @type, fetches its required list, and collects any field not in the block. The printed missing list names exactly the mandatory fields that are absent.

What You'll Build & Learn

Report which required fields a typed block is missing via a lookup table.

- A dict maps types to required fields
- Look up requirements by @type
- Comprehension finds absent fields

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 314: serialize json-ld as a pretty string. Project 316 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a missing required property is the surface task; the transferable Python move is to mark setup, rule, and output before running.

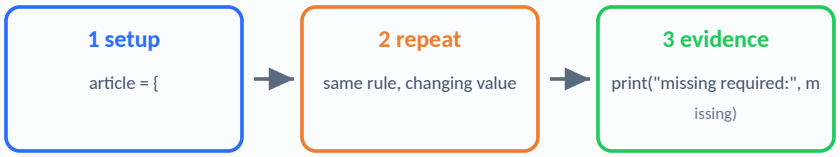
setup -> rule -> result

Logic Lens

- Setup: `article = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("missing required:", missing)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`article = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`missing = [f for f in required[article["@type"]] if f not in article]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print("missing required:", missing)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [f for f in required[article["@type"]] if f not in article]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Missing Required Property: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: article, context, https, schema, org. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: article = { | Core: missing = [f for f in required[article["@type"]] if f not in article] | Evidence: print("missing required:", missing)

🛠️ Reader to Writer Bridge

1. Read: explain detect a missing required property in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 314.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 316 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("missing required:", missing)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `article = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a missing required property and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Indexing the requirements table by the block's own `@type` keeps the rule lookup data-driven, and the comprehension lists each required field the block lacks.

```
article = {
    "@context": "https://schema.org",
    "@type": "Article",
    "author": "Jane Doe",
}
required = {"Article": ["headline", "author", "datePublished"]}
```

```
missing = [f for f in required[article["@type"]] if f not in article]
print("missing required:", missing)
```

```
output
missing required: ['headline', 'datePublished']
```

How it works:

1. The article dict has @context, @type, and author but no headline or datePublished.
2. required maps Article to headline, author, datePublished.
3. The comprehension iterates required[article['@type']] and keeps any field not in article, printing the two missing names.

Try it:

1. Add a Recipe entry to required and test a Recipe block
2. Raise a KeyError-safe message when @type is unknown
3. Return a tuple of present and missing fields
4. Add `headline` to the article and watch the list shrink.

⚠ Common Mistakes

- Hard-coding the Article rules instead of keying by `@type` means the check cannot be reused for other types.
- A KeyError fires if the block's `@type` is not in the requirements table; unknown types need a fallback.
- Listing fields present rather than absent inverts the report.

Skills gained dict lookup | list comprehension | required fields | @type

→ Where You'll Use This: A reusable validator keys its required-field rules off @type, so one function can audit Articles, Products, and Recipes by consulting the matching entry in the table.

⚡ Mental Model

Like a clerk pulling the right form for your visa type, then ticking which mandatory boxes you left blank.

Project 316 Detect a Missing Recommended Property

💡 The Idea Behind It

Beyond mandatory fields, schemas list recommended ones that unlock richer displays such as images, ratings, and brand. This code defines a recommended list, then uses sorted around a comprehension to collect, alphabetically, the recommended fields the product lacks. The output guides incremental improvement: add these to upgrade an eligible block into a fuller result.

What You'll Build & Learn

List the recommended Product fields that are absent, sorted alphabetically.

- Recommended fields differ from required
- They unlock richer results
- sorted orders the missing names

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 315: detect a missing required property. Project 317 changes the surface problem while preserving the underlying method.

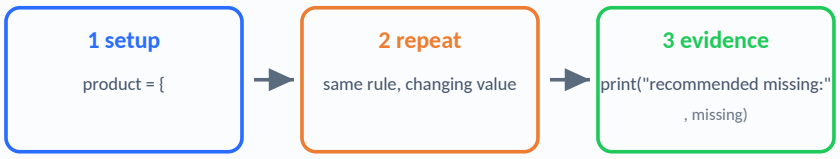
progress check

Reading target: detect a missing recommended property is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `product = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("recommended missing:", missing)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`product = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus:

Move	Code to inspect	Why it matters
		generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`missing = sorted(f for f in recommended if f not in product)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print("recommended missing:", missing)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = sorted(f for f in recommended if f not in product)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Missing Recommended Property: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: product, type, Product, name, Widget. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: product = { | Core: missing = sorted(f for f in recommended if f not in product) | Evidence: print("recommended missing:", missing)

Reader to Writer Bridge

1. Read: explain detect a missing recommended property in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 315.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 317 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("recommended missing:", missing)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `product = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a missing recommended property and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Recommended fields are non-blocking, so they are reported separately from required ones; sorting the missing list keeps the audit output deterministic.

```
product = {
    "@type": "Product",
    "name": "Widget",
    "offers": {"price": "5.00", "priceCurrency": "USD"},
}
recommended = ["image", "description", "brand", "aggregateRating"]
missing = sorted(f for f in recommended if f not in product)
print("recommended missing:", missing)
```

```
output
recommended missing: ['aggregateRating', 'brand', 'description', 'image']
```

How it works:

1. The product dict has name and a minimal offers dict.
2. recommended lists image, description, brand, aggregateRating.
3. sorted wraps a comprehension keeping each recommended field not in product, printing the four absent names alphabetically.

Try it:

1. Weight the recommendations and report the highest-impact missing field first
2. Combine with a required check into one report
3. Count how many recommended fields are present versus absent
4. Add an `image` to the product and watch it drop from the list.

⚠ Common Mistakes

- Treating recommended fields as required would wrongly fail an otherwise valid block.
- Forgetting to sort the comprehension lets the printed order vary across runs.
- Checking nested fields like `offers.priceValidUntil` with a flat `in` test misses them.

Skills gained sorted | recommended fields | list comprehension | membership test

→ Where You'll Use This: After a block passes required-field validation you run this to produce a prioritized to-do list, telling content teams which optional fields would earn star ratings or image enhancements.

Mental Model

Required fields get you in the door; recommended fields are the upgrades that get you the corner office, and this lists which upgrades you skipped.

Project 317 Validate an AggregateRating Value

The Idea Behind It

An AggregateRating must sit within its declared scale or the star widget is rejected. This snippet pulls ratingValue, then checks it lies between worstRating and bestRating using Python's chained comparison. A passing value prints valid; a stray figure like 6 out of 5 would print out of range. It is a one-line bounds guard on the rating.

What You'll Build & Learn

Verify an AggregateRating value falls within its worst-to-best bounds.

- Chained comparison checks a range
- ratingValue must respect worst and best
- Out-of-range stars are rejected

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 316: detect a missing recommended property. Project 318 changes the surface problem while preserving the underlying method.

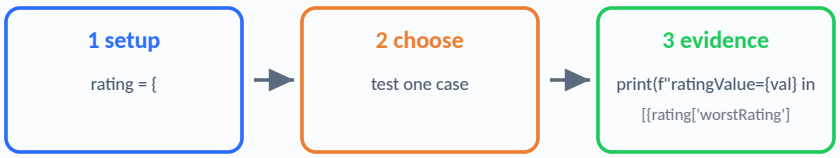
progress check

Reading target: validate an aggregaterating value is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `rating = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"ratingValue={val} in [{rating['worstRating']}, {rating['bestRating']}] -> {'valid' if ok else 'out of range'}")` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`rating = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"ratingValue={val} in [{rating['worstRating']}, {rating['bestRating']}] -> {'valid' if ok else 'out of range'}")`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(f"ratingValue={val} in [{rating['worstRating']}, {rating['bestRating']}] -> {'valid' if ok else 'out of range'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
print(f"ratingValue={val} in [{rating['worstRating']},{rating['bestRating']}] ->
{'valid' if ok else 'out of range'}")
```

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Validate an AggregateRating Value: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: rating, type, AggregateRating, ratingValue, bestRating. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: rating = { | Core: print(f"ratingValue={val} in [{rating['worstRating']},{rating['bestRati... | Evidence: print(f"ratingValue={val} in [{rating['worstRating']},{rating['bestRati...

Reader to Writer Bridge

1. Read: explain validate an aggregaterating value in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 316.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 318 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"ratingValue={val} in [{rating['worstRating']},{rating['bestRating']}] -> {'valid' if ok else 'out of range'}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `rating = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate an aggregaterating value and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Python's chained comparison `worst <= val <= best` reads like the math interval and confirms the rating falls inside the block's own declared bounds.

```
rating = {
    "@type": "AggregateRating",
    "ratingValue": 4.6,
    "bestRating": 5,
    "worstRating": 1,
    "ratingCount": 128,
}
val = rating["ratingValue"]
ok = rating["worstRating"] <= val <= rating["bestRating"]
print(f"ratingValue={val} in [{rating['worstRating']},{rating['bestRating']}] -
> {'valid' if ok else 'out of range'}")
```

```
output
ratingValue=4.6 in [1,5] -> valid
```

How it works:

1. The rating dict holds ratingValue 4.6, bestRating 5, worstRating 1, and ratingCount.
2. The short name val is bound to ratingValue so the comparison reads cleanly.
3. ok uses the chained comparison worstRating <= val <= bestRating, and the f-string prints the value, the bounds, and a valid or out-of-range verdict.

Try it:

1. Default worstRating to 1 when the field is absent
2. Also assert ratingCount is a positive integer
3. Loop the check over a list of rating dicts
4. Set `ratingValue` to 5.5 and watch it fail.

⚠ Common Mistakes

- Assuming a fixed 1-5 scale instead of reading the block's own `bestRating`/`worstRating` misjudges custom scales.
- Comparing strings (`'4.6' <= '5'`) instead of numbers gives lexical, not numeric, results.
- Forgetting that `ratingValue` of exactly `bestRating` is valid (inclusive bound).

Skills gained chained comparison | AggregateRating | range check | f-strings

→ Where You'll Use This: Review-aggregation pages compute an average star rating from many reviews; this guards against an arithmetic slip producing a value outside the declared scale and losing the star snippet.

Mental Model

Like a thermometer that must read between freezing and boiling; a value off the dial means the instrument is broken and gets ignored.

Project 318 Validate an Offer Availability Enum

The Idea Behind It

The Offer availability field must be one of schema.org's fixed enum URLs, never free text. This code defines the allowed set of availability URLs, then loops over a few offers checking each value's membership. It also splits the URL on the last slash to print a short label. A made-up value like SoldOut fails the set check and prints invalid enum.

What You'll Build & Learn

Validate each Offer availability against the allowed schema.org enum set.

- Availability is a fixed enum of URLs
- Set membership validates the value
- rsplit extracts the short label

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 317: validate an aggregaterating value. Project 319 changes the surface problem while preserving the underlying method.

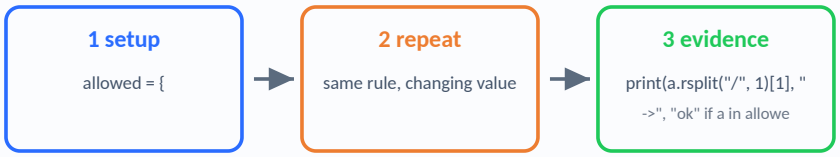
progress check

Reading target: validate an offer availability enum is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `allowed = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enum")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`allowed = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enum")`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enum")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enum")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate an Offer Availability Enum: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: allowed, https, schema, org, InStock. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: allowed = { | Core: print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enu... | Evidence: print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enu...

🛠️ Reader to Writer Bridge

1. Read: explain validate an offer availability enum in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 317.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 319 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enum")`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `allowed = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate an offer availability enum and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

An enum field has a closed set of legal values; a set-membership test is the validation, and `rsplit('/', 1)[1]` peels off the short label for a readable report.

```
allowed = {
    "https://schema.org/InStock",
    "https://schema.org/OutOfStock",
    "https://schema.org/PreOrder",
    "https://schema.org/Discontinued",
}
offers = [
    {"availability": "https://schema.org/InStock"},
    {"availability": "https://schema.org/SoldOut"},
]
for o in offers:
    a = o["availability"]
    print(a.rsplit("/", 1)[1], "->", "ok" if a in allowed else "invalid enum")
```

```
output
InStock -> ok
SoldOut -> invalid enum
```

How it works:

1. allowed is a set of the four valid availability URLs.
2. The loop reads each offer's availability into a.
3. a.rsplit('/', 1)[1] prints the trailing label, and a in allowed decides ok or invalid enum, catching SoldOut.

Try it:

1. Map common phrases like 'sold out' to the correct enum URL
2. Add LimitedAvailability and BackOrder to the allowed set
3. Report all invalid offers as a list at the end
4. Add `BackOrder` to the allowed set and test it.

⚠ Common Mistakes

- Matching on the bare label `InStock` instead of the full enum URL accepts an unqualified value the schema rejects.

- Using `OutOfStock` spelling drift like `SoldOut` is a frequent real mistake this catches.
- Splitting with `split('/')[-1]` works too, but assuming a fixed index breaks on a trailing slash.

Skills gained set membership | rsplit | enum validation | loop

→ Where You'll Use This: When generating product markup from a catalog feed, the stock status often arrives as words like sold out; this validation forces it onto the canonical schema.org URLs Google accepts.

⚡ Mental Model

Like a drop-down menu with fixed choices: you may pick InStock or OutOfStock, but typing your own option simply does not register.

Project 319 Nest an Author as a Person

💡 The Idea Behind It

An author can be a bare string, but a nested Person object is richer because it can carry a URL and tie the byline to an entity. This code sets the Article's author to a Person dict with name and url, then reads back author['@type'] and author['name']. The byline becomes a typed sub-dict rather than loose text.

What You'll Build & Learn

Express an Article author as a nested Person with a linking URL.

- Author may be a string or a Person
- A Person can carry a url
- The author value becomes a typed sub-dict

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 318: validate an offer availability enum. Project 320 changes the surface problem while preserving the underlying method.

progress check

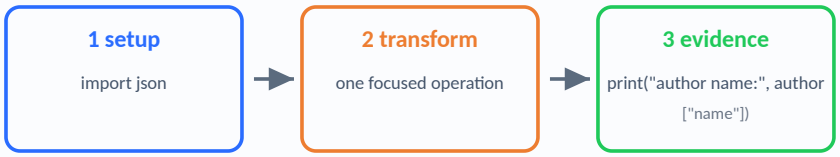
Reading target: nest an author as a person is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.

- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("author name:", author["name"])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print("author name:", author["name"])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Nest an Author as a Person: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, article, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print("author name:", author["name"])

🛠️ Reader to Writer Bridge

1. Read: explain nest an author as a person in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 318.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 320 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("author name:", author["name"])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of nest an author as a person and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Replacing a string author with a `Person` dict lets the byline carry its own `@type`, name, and profile URL, which strengthens entity association.

```python
import json
article = {
    "@context": "https://schema.org",
    "@type": "Article",
    "headline": "Inside the Index",
    "author": {
        "@type": "Person",
        "name": "Sam Lee",
        "url": "https://example.com/sam",
    },
}
author = article["author"]
print("author type:", author["@type"])
print("author name:", author["name"])
```

```
output
author type: Person
author name: Sam Lee
```

How it works:

1. The article dict declares Article with a headline.
2. Its author key holds a nested Person dict with @type Person, name, and url.
3. author aliases that sub-dict, and the prints show the author type Person and the name Sam Lee.

Try it:

1. Add a sameAs list to the Person linking to social profiles
2. Support a list of two co-authors
3. Fall back to a string author when no url is available
4. Change the author to an `Organization` instead of a `Person`.

⚠ Common Mistakes

- Reading `article['author']` as a string and calling `.upper()` breaks once the author is a dict.
- Omitting `@type: Person` on the nested object leaves the author untyped.
- Putting the author URL on the Article instead of inside the Person mislinks it.

Skills gained nested Person | entity linking | dict access | Article schema

→ Where You'll Use This: Author bio and article pages use a Person author with a url to a profile, strengthening author authority signals and connecting the byline across a site's content.

Mental Model

A string byline is a printed name; a Person object is a name badge with a link back to the person's full file.

Project 320 Validate a Review Rating Is 1 to 5

The Idea Behind It

A Review's rating normally lives on a 1-to-5 star scale, so a batch audit flags anything outside that band. This code loops over a list of review dicts, reads each ratingValue into v, and tests 1 <= v <= 5 with a chained comparison. A zero or a six prints out of 1-5 while valid scores print ok, catching corrupt ratings before they ship.

What You'll Build & Learn

Scan a batch of reviews and flag ratings outside the 1-to-5 band.

- Reviews use a 1-to-5 star scale
- Chained comparison bounds the value
- Loop audits a batch

Skill Roadmap

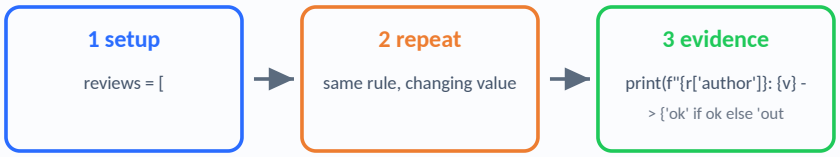
Skill step: moves you from recognition toward independent tracing. It builds on Project 319: nest an author as a person. Project 321 changes the surface problem while preserving the underlying method. *progress check*

Reading target: validate a review rating is 1 to 5 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `reviews = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1–5'}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`reviews = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1–5'}")`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1–5'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1–5'}")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate a Review Rating Is 1 to 5: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: reviews, author, ratingValue, ok, of. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: reviews = [| Core: print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1-5'}") | Evidence: print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1-5'}")

🛠️ Reader to Writer Bridge

1. Read: explain validate a review rating is 1 to 5 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 319.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 321 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1–5'}")`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `reviews = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate a review rating is 1 to 5 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A chained `1 <= v <= 5` comparison per review separates valid star values from impossible ones such as a zero rating.

```
reviews = [
    {"author": "A", "ratingValue": 5},
    {"author": "B", "ratingValue": 0},
    {"author": "C", "ratingValue": 3},
]
for r in reviews:
```

```
    v = r["ratingValue"]
    ok = 1 <= v <= 5
    print(f"{r['author']}: {v} -> {'ok' if ok else 'out of 1-5'}")
```

```
output
A: 5 -> ok
B: 0 -> out of 1-5
C: 3 -> ok
```

How it works:

1. reviews is a list of three dicts each with an author and a ratingValue.
2. The loop reads v from each review.
3. ok uses 1 <= v <= 5, and the f-string prints the author, value, and ok or out of 1-5, flagging the zero.

Try it:

1. Collect the offending authors into a list instead of printing inline
2. Allow a configurable max so a 1-to-10 scale also works
3. Skip reviews missing the ratingValue key without crashing
4. Add a review with rating 4.5 and confirm fractional values pass.

⚠ Common Mistakes

- Allowing 0 because a falsy check (`if v`) treats zero specially instead of using a range test.
- Hard-coding 1-5 when a block declares its own `bestRating` of, say, 10.
- Comparing a string rating to integers without converting first.

Skills gained chained comparison | loop | Review rating | batch validation

→ Where You'll Use This: When importing reviews from a third-party feed you run this gate so a malformed rating, like a zero placeholder, never reaches the markup and triggers a structured-data error.

⚡ Mental Model

Like a referee checking every scorecard reads between one and five stars; a zero or a six means someone mis-marked the card.

Project 321 Check a Type Against a Schema Dict

The Idea Behind It

A data-driven validator stores each type's required fields in one table and validates any block by looking up its @type. This code defines SCHEMAS mapping three types to their requirements, reads the document's type,

fetches the matching list, and collects missing fields with a comprehension. Adding a new type means adding a table row, so the design scales cleanly.

What You'll Build & Learn

Validate any typed block against a central schema-to-fields table.

- One table holds many types' requirements
- Look up requirements by @type
- Comprehension finds missing fields

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 320: validate a review rating is 1 to 5. Project 322 changes the surface problem while preserving the underlying method.

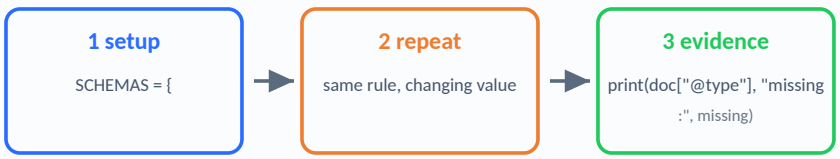
progress check

Reading target: check a type against a schema dict is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `SCHEMAS = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(doc["@type"], "missing:", missing)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`SCHEMAS = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`missing = [f for f in req if f not in doc]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(doc["@type"], "missing:", missing)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [f for f in req if f not in doc]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check a Type Against a Schema Dict: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: Article, headline, author, datePublished, Product. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: SCHEMAS = { | Core: missing = [f for f in req if f not in doc] | Evidence: print(doc["@type"], "missing:", missing)

Reader to Writer Bridge

1. Read: explain check a type against a schema dict in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 320.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 322 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(doc["@type"], "missing:", missing)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `SCHEMAS = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check a type against a schema dict and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Keying the requirements table by `@type` lets one comprehension validate Article, Product or Recipe blocks identically, the heart of a generic schema linter.

```
SCHEMAS = {
    "Article": ["headline", "author", "datePublished"],
    "Product": ["name", "offers"],
    "Recipe": ["name", "recipeIngredient", "recipeInstructions"],
}
doc = {"@type": "Recipe", "name": "Soup", "recipeIngredient": ["water"]}
req = SCHEMAS[doc["@type"]]
missing = [f for f in req if f not in doc]
print(doc["@type"], "missing:", missing)
```

```
output
Recipe missing: ['recipeInstructions']
```

How it works:

1. SCHEMAS maps Article, Product, and Recipe to their required field lists.
2. The doc declares Recipe with name and one ingredient but no instructions.
3. req fetches SCHEMAS[doc['@type']] and the comprehension prints the single missing field recipeInstructions.

Try it:

1. Add Event and LocalBusiness rows to SCHEMAS
2. Handle an unknown @type gracefully with dict.get
3. Load the SCHEMAS table from an external JSON file
4. Add the missing `recipeInstructions` and watch the list empty.

⚠ Common Mistakes

- A block whose `@type` is not a key in `SCHEMAS` raises KeyError; unknown types need an explicit branch.
- Storing required fields as a comma string instead of a list breaks the `in` iteration.
- Mutating `req` (the shared list) inside the loop would corrupt the table for later blocks.

Skills gained dict-driven design | list comprehension | schema table | @type lookup

→ Where You'll Use This: A production structured-data linter is built exactly this way: a registry of types and their rules so one engine validates an entire site's mixed markup from a single table.

Mental Model

Like a rulebook indexed by category: look up the entry for your kind of thing and the required checklist comes back, no special-casing per type.

Project 322 Check Rich-Result Eligibility

The Idea Behind It

A block earns a rich result only when it has every required field for its type. This code stores Product's requirements, then iterates a small catalog computing each entry's missing fields. A product with no gaps prints eligible; one lacking image prints needs ['image']. It scales the single-block check across many products in one loop.

What You'll Build & Learn

Report rich-result eligibility for each product in a small catalog.

- Eligibility means no missing required fields
- Loop applies the check per product
- Report the specific gaps

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 321: check a type against a schema dict. Project 323 changes the surface problem while preserving the underlying method.

progress check

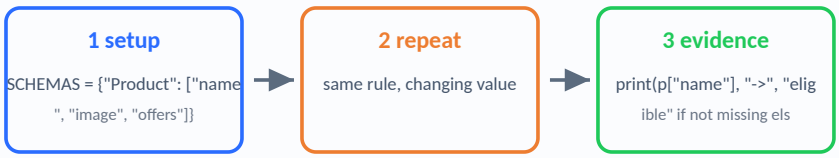
Reading target: check rich-result eligibility is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `SCHEMAS = {"Product": ["name", "image", "offers"]}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(p["name"], "->", "eligible" if not missing else f"needs {missing}")` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`SCHEMAS = {"Product": ["name", "image", "offers"]}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`missing = [f for f in req if f not in p]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(p["name"], "->", "eligible" if not missing else f"needs {missing}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [f for f in req if f not in p]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Rich-Result Eligibility: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: Product, name, image, offers, products. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: SCHEMAS = {"Product": ["name", "image", "offers"]} | Core: missing = [f for f in req if f not in p] | Evidence: print(p["name"], "->", "eligible" if not missing else f"needs {missing}...

Reader to Writer Bridge

1. Read: explain check rich-result eligibility in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 321.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 323 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(p["name"], "->", "eligible" if not missing else f"needs {missing}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `SCHEMAS = {"Product": ["name", "image", "offers"]}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check rich-result eligibility and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Eligibility is just an empty missing-field list; reporting the gap for ineligible products tells the author exactly what to add.

```
SCHEMAS = {"Product": ["name", "image", "offers"]}
products = [
    {"@type": "Product", "name": "A", "image": "a.jpg", "offers": {}},
    {"@type": "Product", "name": "B", "offers": {}},
]
req = SCHEMAS["Product"]
for p in products:
    missing = [f for f in req if f not in p]
    print(p["name"], "->", "eligible" if not missing else f"needs {missing}")
```

```
output
A -> eligible
B -> needs ['image']
```

How it works:

1. SCHEMAS maps Product to name, image, offers.
2. products is a two-item list; A has all three, B lacks image.
3. The loop computes missing per product and prints eligible or needs plus the missing list, marking B as needing image.

Try it:

1. Count how many products are eligible overall
2. Write the failing products to a separate report list
3. Add a second type and dispatch each block by its @type
4. Give product B an `image` and watch it become eligible.

⚠ Common Mistakes

- Treating an empty `offers` dict as missing would wrongly disqualify product A; presence of the key, not its contents, is checked here.
- Calling a product eligible when it merely has the keys but with empty values is a deeper trap to watch for.
- Sharing the `req` list and appending to it inside the loop pollutes later iterations.

Skills gained loop eligibility check list comprehension catalog audit

→ **Where You'll Use This:** Before a deploy you sweep the whole product catalog and get a list of which pages will earn price snippets and which need a field added, turning validation into an action list.

Mental Model

Like a bouncer with a guest list: each product is checked against the requirements and either waved through or told what credential it is missing.

Project 323 Detect a Wrong @type Value

The Idea Behind It

A typo in @type silently disables structured data because engines ignore types they do not recognize. This code holds a set of known type names, then loops over documents checking each @type for membership. Article passes as known while Artical and Prodcut, simple misspellings, print unknown type, surfacing errors that would otherwise vanish without warning.

What You'll Build & Learn

Catch misspelled @type values by checking against a known-types set.

- Unknown types are silently ignored
- A known-types set catches typos
- Set membership is the test

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 322: check rich-result eligibility. Project 324 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect a wrong @type value is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

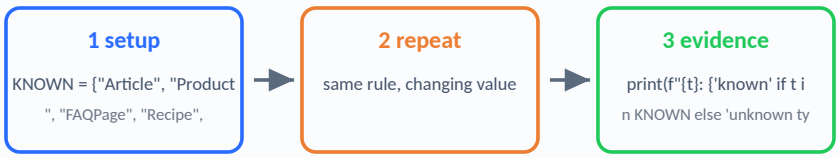
Logic Lens

- Setup: `KNOWN = {"Article", "Product", "FAQPage", "Recipe", "Event", "LocalBusiness"}` gives the program its first fact, tool, or entry point. Use it as the anchor for the

later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`KNOWN = {"Article", "Product", "FAQPage", "Recipe", "Event", "LocalBusiness"}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Wrong @type Value: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: Article, Product, FAQPage, Recipe, Event. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: KNOWN = {"Article", "Product", "FAQPage", "Recipe", "Event", "LocalBusi... | Core: print(f"{t}: {'known' if t in KNOWN else 'unknown type'}") | Evidence: print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")

🛠️ Reader to Writer Bridge

1. Read: explain detect a wrong @type value in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 322.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 324 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `KNOWN = {"Article", "Product", "FAQPage", "Recipe", "Event", "LocalBusiness"}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a wrong @type value and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Validating the `@type` against a curated set of known types catches the silent failures that come from misspelling a schema name.

```
KNOWN = {"Article", "Product", "FAQPage", "Recipe", "Event", "LocalBusiness"}
docs = [
    {"@type": "Article"},
    {"@type": "Artical"},
    {"@type": "Prodcut"},
]
for d in docs:
    t = d["@type"]
    print(f"{t}: {'known' if t in KNOWN else 'unknown type'}")
```

```
output
Article: known
Artical: unknown type
Prodcut: unknown type
```

How it works:

1. KNOWN is a set of six valid schema type names.
2. docs holds three blocks: Article plus the typos Artical and Prodcut.
3. The loop reads each @type into t and prints known or unknown type via t in KNOWN, flagging both misspellings.

Try it:

1. Suggest the closest known type using difflib for a near miss
2. Pull KNOWN from a complete list of schema.org types
3. Also flag a block missing the @type key entirely
4. Add `WebPage` to the known set and test it.

⚠ Common Mistakes

- Case-folding mismatches: `article` (lowercase) would be flagged unknown because schema types are PascalCase.
- Maintaining the known-types set by hand means a real new type can be wrongly rejected.

- Checking only that `@type` exists, not that its value is recognised, misses typos entirely.

Skills gained set membership | @type validation | loop | typo detection

→ Where You'll Use This: Hand-edited or template-generated markup easily introduces a type typo that breaks rich results invisibly; this check is a cheap guard that runs across every page's @type.

Mental Model

Like spell-checking a destination on a parcel: Artical is not on the route map, so the package quietly goes nowhere.

Project 324 Validate a Date Field With strptime

The Idea Behind It

Schema date fields must be ISO-8601 YYYY-MM-DD, and parsing is the cleanest way to validate them. This code loops over candidate strings inside a try block, calling datetime.strptime with the ISO format. A successful parse means valid ISO; a ValueError is caught and reported as bad format, so an impossible date or a slashed US format both fail clearly.

What You'll Build & Learn

Validate date strings as ISO-8601 by parsing with strptime.

- Schema dates use YYYY-MM-DD
- strptime parses against a format
- A ValueError signals a bad date

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 323: detect a wrong @type value. Project 325 changes the surface problem while preserving the underlying method.

progress check

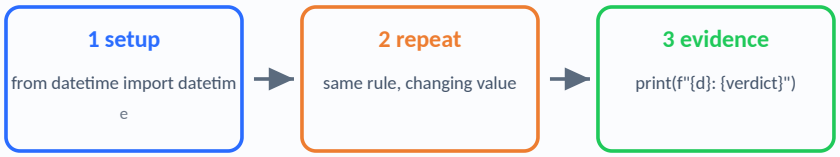
Reading target: validate a date field with strptime is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from datetime import datetime` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"{d}: {verdict}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from datetime import datetime`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`for d in dates:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{d}: {verdict}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for d in dates:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Read the try path as the hoped-for route and the catch/except/finally path as the boundary plan.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate a Date Field With strptime: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: datetime, dates, strptime, verdict, valid. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from datetime import datetime | Core: for d in dates: | Evidence: print(f"{d}: {verdict}")

🛠️ Reader to Writer Bridge

1. Read: explain validate a date field with strptime in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 323.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 325 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{d}: {verdict}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from datetime import datetime` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of validate a date field with strptime and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`strptime` with the `%Y-%m-%d` pattern both checks the shape and the real calendar bounds, so month 13 and a US-style slash date are both rejected.

```
from datetime import datetime
dates = ["2026-01-15", "2026-13-40", "01/15/2026"]
for d in dates:
    try:
        datetime.strptime(d, "%Y-%m-%d")
        verdict = "valid ISO"
    except ValueError:
        verdict = "bad format"
    print(f"{d}: {verdict}")
```

```
output
2026-01-15: valid ISO
2026-13-40: bad format
01/15/2026: bad format
```

How it works:

1. dates lists a valid date, an impossible 2026-13-40, and a US-style 01/15/2026.
2. The loop tries datetime.strptime(d, '%Y-%m-%d') inside a try.
3. A clean parse sets valid ISO; the except ValueError sets bad format, so only the first date passes.

Try it:

1. Add ISO date-time support by trying a second format on failure
2. Return the parsed datetime object on success
3. Reject future dates for a datePublished field
4. Accept a full datetime by switching the pattern to include `T%H:%M`.

⚠ Common Mistakes

- Catching a bare `except` instead of `ValueError` would also swallow unrelated bugs.
- A loose regex like `\d{4}-\d{2}-\d{2}` would accept the impossible `2026-13-40` that strptime correctly rejects.
- Relying on `datetime.now` for a reference date is forbidden and unnecessary; format validation needs no clock.

Skills gained `strptime` `try/except` `ISO-8601 date` `ValueError`

→ **Where You'll Use This:** datePublished, uploadDate, and startDate must all parse as ISO dates; this validation catches a content team pasting a localized or impossible date that would invalidate the block.

Mental Model

Like handing a date to a strict clerk who only accepts one form; if they can fill it in, it is valid, if they choke, it is rejected.

Project 325 Find Gaps in Breadcrumb Positions

The Idea Behind It

A breadcrumb numbered 1, 2, 4 has a hole that breaks the chain. This code sorts the present positions, builds the full expected range up to the maximum as a set, and subtracts to find the gaps. The set difference pinpoints the missing 3. It detects skips that simple ordering checks would miss because the sequence still rises.

What You'll Build & Learn

Find skipped positions in a breadcrumb using set difference.

- Set difference finds missing numbers
- range to the max builds the full set
- Sorting normalizes the input

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 324: validate a date field with strptime. Project 326 changes the surface problem while preserving the underlying method.

progress check

Reading target: find gaps in breadcrumb positions is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `items = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.

- Evidence: `print("gaps:", gaps if gaps else "none")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`items = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print("gaps:", gaps if gaps else "none")`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print("gaps:", gaps if gaps else "none")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print("gaps:", gaps if gaps else "none")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Gaps in Breadcrumb Positions: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: items, position, name, Home, Blog. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: items = [| Core: print("gaps:", gaps if gaps else "none") | Evidence: print("gaps:", gaps if gaps else "none")

🛠️ Reader to Writer Bridge

1. Read: explain find gaps in breadcrumb positions in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 324.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 326 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("gaps:", gaps if gaps else "none")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `items = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find gaps in breadcrumb positions and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Building the full expected set from 1 to the maximum position and subtracting the present positions leaves exactly the missing numbers.

```
items = [
    {"position": 1, "name": "Home"},
    {"position": 2, "name": "Blog"},
    {"position": 4, "name": "Post"},
]
```

```
positions = sorted(it["position"] for it in items)
expected = set(range(1, max(positions) + 1))
gaps = sorted(expected - set(positions))
print("positions:", positions)
print("gaps:", gaps if gaps else "none")
```

```
output
positions: [1, 2, 4]
gaps: [3]
```

How it works:

1. items holds positions 1, 2, 4.
2. positions sorts the values and expected is set(range(1, max+1)).
3. gaps is sorted(expected - set(positions)), printing the present positions and the missing 3.

Try it:

1. Also report duplicate positions using a Counter
2. Detect gaps when positions do not start at 1
3. Return the gaps from a function for use in a larger linter
4. Add the missing position-3 item and watch gaps clear.

⚠ Common Mistakes

- Using `len(items)` instead of `max(positions)` as the range top hides a gap that pushes the max beyond the count.
- Forgetting to sort the gap set makes the printed order nondeterministic.
- Subtracting in the wrong direction (`present - expected`) reports extras, not gaps.

Skills gained set difference | range | sorted | gap detection

→ Where You'll Use This: When a CMS drops a middle breadcrumb level, the positions still ascend so a strict ordering check passes; this set-based audit is what actually catches the missing rung.

Mental Model

Like checking a numbered fence for missing pickets: lay out every slot from one to the highest, then see which numbers have no picket.

Project 326 Combine Multiple Types in a Graph

The Idea Behind It

A single page often describes several entities at once, and the @graph array lets one block hold many typed nodes under one shared context. This code builds a graph dict with @context plus an @graph list of Organization, WebSite, and BreadcrumbList nodes. It collects each node's @type with a comprehension and reports the node count, showing how multiple entities coexist.

What You'll Build & Learn

Combine several typed entities into one block using @graph.

- @graph holds multiple typed nodes
- One @context covers them all
- Comprehension lists the node types

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 325: find gaps in breadcrumb positions. Project 327 changes the surface problem while preserving the underlying method.

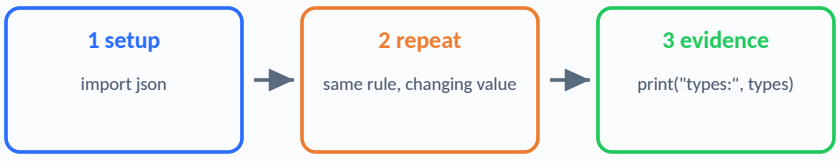
progress check

Reading target: combine multiple types in a graph is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("types:", types)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`types = [node["@type"] for node in graph["@graph"]]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print("types:", types)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `types = [node["@type"] for node in graph["@graph"]]`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Combine Multiple Types in a Graph: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, graph, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.

Layer	What to read
Evidence	Entry: import json | Core: types = [node["@type"] for node in graph["@graph"]] | Evidence: print("types:", types)

🛠️ Reader to Writer Bridge

1. Read: explain combine multiple types in a graph in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 325.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 327 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("types:", types)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of combine multiple types in a graph and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`@graph` is an array of independent typed nodes sharing one `@context`, the idiomatic way to declare multiple entities on a page without repeating the context.

```python
import json
graph = {
    "@context": "https://schema.org",
    "@graph": [
        {"@type": "Organization", "name": "Acme"},
        {"@type": "WebSite", "name": "Acme Blog"},
        {"@type": "BreadcrumbList", "itemListElement": []},
    ],
}
types = [node["@type"] for node in graph["@graph"]]
print("nodes:", len(graph["@graph"]))
print("types:", types)
```

```
output
```

```
nodes: 3
types: ['Organization', 'WebSite', 'BreadcrumbList']
```

How it works:

1. The graph dict has @context and an @graph list of three nodes.
2. The nodes declare Organization, WebSite, and BreadcrumbList.
3. types collects each node's @type via a comprehension, and the prints report 3 nodes and the three type names.

Try it:

1. Add an @id to each node so they can cross-reference
2. Append a WebPage node to the graph
3. Count how many distinct types the graph contains using a set
4. Add a `WebPage` node and watch the count rise.

⚠ Common Mistakes

- Giving each node its own `@context` inside `@graph` is redundant and bloats the block.
- Reading `graph['@type']` fails because the top-level object has none; the types live on the nodes.
- Mixing a single-entity layout with a `@graph` layout on the same page confuses parsers.

Skills gained `@graph` `list comprehension` `multiple entities` `shared context`

→ Where You'll Use This: Real homepages emit one @graph describing the organization, the site, the search box, and breadcrumbs together, which is the recommended way to package related markup on a page.

⚡ Mental Model

Like one envelope holding several documents that all share the same letterhead; @context is the letterhead, @graph is the stack inside.

Project 327 Build a HowTo With Steps

💡 The Idea Behind It

A HowTo turns a tutorial into numbered, individually displayable steps in search. This code builds the howto dict with a step list of HowToStep objects, each carrying its own text. Looping with enumerate starting at 1 restores the human numbering and prints Step 1, Step 2, Step 3, mirroring how the engine renders each step as a distinct instruction.

What You'll Build & Learn

Build a HowTo whose ordered HowToStep objects print as numbered steps.

- HowTo holds a step list
- Each step is a HowToStep with text
- enumerate from 1 restores numbering

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 326: combine multiple types in a graph. Project 328 changes the surface problem while preserving the underlying method.

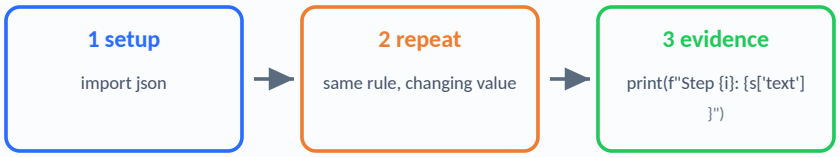
progress check

Reading target: build a howto with steps is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"Step {i}: {s['text']}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.

Move	Code to inspect	Why it matters
2. Apply the rule	`for i, s in enumerate(howto["step"], 1):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"Step {i}: {s['text']}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for i, s in enumerate(howto["step"], 1):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a HowTo With Steps: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, howto, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: for i, s in enumerate(howto["step"], 1): | Evidence: print(f"Step {i}: {s['text']}")

Reader to Writer Bridge

1. Read: explain build a howto with steps in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 326.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 328 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"Step {i}: {s['text']}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a howto with steps and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The `step` array holds ordered `HowToStep` objects; `enumerate(..., 1)` numbers them for display while the array order itself defines the sequence.

```
import json
howto = {
    "@context": "https://schema.org",
    "@type": "HowTo",
    "name": "Change a Tire",
    "step": [
        {"@type": "HowToStep", "text": "Loosen the lug nuts."},
        {"@type": "HowToStep", "text": "Jack up the car."},
        {"@type": "HowToStep", "text": "Swap the tire."},
    ],
}
for i, s in enumerate(howto["step"], 1):
    print(f"Step {i}: {s['text']}")
```

```
output
Step 1: Loosen the lug nuts.
Step 2: Jack up the car.
Step 3: Swap the tire.
```

How it works:

1. The howto dict declares HowTo with a name.
2. step is a list of three HowToStep dicts, each with a text instruction.
3. The loop uses enumerate(howto['step'], 1) and prints Step i and the step text for all three.

Try it:

1. Add an image to each HowToStep
2. Add a tool or supply list to the HowTo
3. Wrap groups of steps in HowToSection objects
4. Add a fourth step and confirm the numbering continues.

⚠ Common Mistakes

- Using the key `steps` (plural) instead of the schema's `step` breaks the HowTo.
- Relying on a `position` field for ordering when HowTo derives order from array index instead.
- Storing steps as raw strings loses the per-step `@type` that lets each render separately.

Skills gained HowTo schema | HowToStep | enumerate | loop

→ Where You'll Use This: Tutorial and instructional pages markup their procedures as HowTo so search can present the steps individually, sometimes with a step-by-step visual carousel.

Mental Model

Like an IKEA instruction sheet: each panel is a self-contained step, numbered so you know the order to follow.

Project 328 Build a VideoObject Entity

The Idea Behind It

A VideoObject lets a video appear as a rich result with a thumbnail and upload date. This code builds the video dict with name, description, thumbnailUrl, uploadDate, and duration, then validates by listing the four required fields and collecting any not in the dict. Since all four are present, the audit prints none, confirming eligibility.

What You'll Build & Learn

Build a VideoObject and confirm its four required fields are present.

- VideoObject enables a video rich result
- Required: name, description, thumbnailUrl, uploadDate
- Comprehension audits presence

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 327: build a howto with steps. Project 329 changes the surface problem while preserving the underlying method. *progress check*

Reading target: build a videoobject entity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("VideoObject missing:", missing if missing else "none")` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`missing = [f for f in required if f not in video]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print("VideoObject missing:", missing if missing else "none")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [f for f in required if f not in video]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a VideoObject Entity: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, video, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: missing = [f for f in required if f not in video] | Evidence: print("VideoObject missing:", missing if missing else "none")

🛠️ Reader to Writer Bridge

1. Read: explain build a videoobject entity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 327.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 329 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("VideoObject missing:", missing if missing else "none")`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a videoobject entity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

VideoObject has four required fields beyond context and type; listing any that are absent confirms eligibility for a video rich result in one check.

```
import json
video = {
    "@context": "https://schema.org",
    "@type": "VideoObject",
    "name": "Intro to SEO",
    "description": "A short primer.",
    "thumbnailUrl": "https://example.com/thumb.jpg",
    "uploadDate": "2026-02-01",
    "duration": "PT3M30S",
}
required = ["name", "description", "thumbnailUrl", "uploadDate"]
missing = [f for f in required if f not in video]
print("VideoObject missing:", missing if missing else "none")
```

```
output
VideoObject missing: none
```

How it works:

1. The video dict declares VideoObject with name, description, thumbnailUrl, uploadDate, and a duration.
2. required lists the four mandatory fields.
3. The comprehension keeps any required field not in video, and the print reports none since all four exist.

Try it:

1. Add a contentUrl and embedUrl to the block
2. Validate uploadDate parses as an ISO date
3. Add a duration regex check for the ISO-8601 form
4. Remove `description` and watch it appear in the missing list.

⚠ Common Mistakes

- Spelling `thumbnailUrl` as `thumbnailURL` (wrong casing) makes the required field appear missing.
- Providing `duration` but omitting `uploadDate` still fails, since duration is optional and uploadDate is required.

- Writing `duration` as `3:30` rather than the ISO `PT3M30S` breaks any duration-based feature.

Skills gained VideoObject | required fields | list comprehension | audit

→ Where You'll Use This: Pages hosting or embedding video emit VideoObject markup so the clip can show with a thumbnail and date in search and the dedicated video results, driving extra visibility.

⚡ Mental Model

Like a video's poster frame and release date on a streaming tile: the four required fields are the tile's caption and image.

Project 329 Build a Review With an Author

💡 The Idea Behind It

A standalone Review names what it reviews, the star rating given, and who wrote it, each as a nested typed object. This code builds the review dict where itemReviewed is a Product, reviewRating is a Rating with ratingValue and bestRating, and author is a Person. The prints reach into each sub-dict to surface the reviewed item, the score, and the author name.

What You'll Build & Learn

Build a Review linking a reviewed item, a rating, and an author.

- itemReviewed, reviewRating, author all nest
- Each is a typed sub-object
- Reach into sub-dicts to read values

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 328: build a videoobject entity. Project 330 changes the surface problem while preserving the underlying method. *progress check*

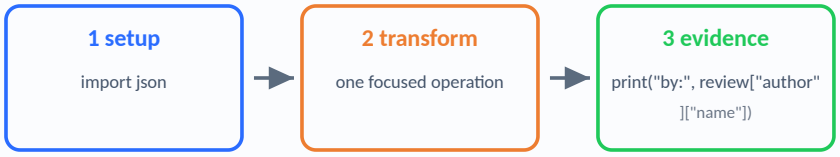
Reading target: build a review with an author is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.

- Evidence: `print("by:", review["author"]["name"])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print("by:", review["author"]["name"])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Build a Review With an Author: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, review, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print("by:", review["author"]["name"])

🛠️ Reader to Writer Bridge

1. Read: explain build a review with an author in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 328.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 330 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("by:", review["author"]["name"])`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a review with an author and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A Review threads together three typed sub-objects: the `itemReviewed` Product, the `reviewRating` Rating, and the `author` Person, each addressed by drilling into nested keys.

```python
import json
review = {
    "@context": "https://schema.org",
    "@type": "Review",
    "itemReviewed": {"@type": "Product", "name": "Blender"},
    "reviewRating": {"@type": "Rating", "ratingValue": 4, "bestRating": 5},
    "author": {"@type": "Person", "name": "Pat Kim"},
}
print("reviewed:", review["itemReviewed"]["name"])
print("rating:", review["reviewRating"]["ratingValue"])
print("by:", review["author"]["name"])
```

```
output
reviewed: Blender
rating: 4
by: Pat Kim
```

How it works:

1. The review dict declares Review.
2. itemReviewed is a nested Product, reviewRating is a Rating with ratingValue 4 and bestRating 5, and author is a Person.
3. The prints read review['itemReviewed']['name'], the ratingValue, and the author name.

Try it:

1. Add a reviewBody field with the prose
2. Add a datePublished to the review
3. Include a worstRating on the Rating and validate the range
4. Add a `reviewBody` with the prose of the review.

⚠ Common Mistakes

- Confusing `reviewRating` (one reviewer's score) with `aggregateRating` (the average of many).
- Putting the rating number directly on the Review instead of inside a `Rating` object.
- Naming the field `itemReviewed` is required; `reviewedItem` will not parse.

Skills gained Review schema | nested objects | Rating | dict access

→ Where You'll Use This: Editorial and product-review pages emit a Review block so the star rating and reviewer can show in search, distinct from aggregated ratings drawn from many reviewers.

⚡ Mental Model

Like a movie review byline: the film reviewed, the stars awarded, and the critic's name, here each filed as its own labeled card.

Project 330 Build a Sitelinks Searchbox

💡 The Idea Behind It

A WebSite block with a SearchAction can grant a search box directly in Google's results. This code builds the site dict whose potentialAction is a SearchAction with a target URL and a query-input spec. The target embeds a {query} placeholder the engine fills with the user's term. The code reads the action type and confirms the placeholder is present.

What You'll Build & Learn

Build a WebSite SearchAction enabling a sitelinks search box.

- potentialAction holds a SearchAction
- target embeds a {query} placeholder
- query-input names the bound parameter

🧭 Skill Roadmap

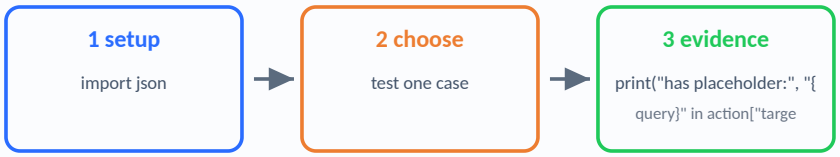
Skill step: pushes the same idea into a less automatic situation. It builds on Project 329: build a review with an author. Project 331 changes the surface problem while preserving the underlying method. *progress check*

Reading target: build a sitelinks searchbox is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("has placeholder:", "{query}" in action["target"])` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. slow line
3. Check the evidence	`print("has placeholder:", "{query}" in action["target"])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Build a Sitelinks Searchbox: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, site, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print("has placeholder:", "{query}" in action["target"])

🛠️ Reader to Writer Bridge

1. Read: explain build a sitelinks searchbox in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 329.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 331 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("has placeholder:", "{query}" in action["target"])`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a sitelinks searchbox and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The `potentialAction` is a `SearchAction` whose `target` must contain a `{query}` token matching the name declared in `query-input`, wiring the results-page search box to your site search.

```
import json
site = {
    "@context": "https://schema.org",
    "@type": "WebSite",
    "url": "https://example.com",
    "potentialAction": {
```

```
        "@type": "SearchAction",
        "target": "https://example.com/search?q={query}",
        "query-input": "required name=query",
    },
}
action = site["potentialAction"]
print("action:", action["@type"])
print("has placeholder:", "{query}" in action["target"])
```

```
output
action: SearchAction
has placeholder: True
```

How it works:

1. The site dict declares WebSite with a url.
2. potentialAction is a SearchAction with a target URL containing {query} and a query-input string.
3. action aliases it, printing the action type SearchAction and confirming {query} is in the target via membership.

Try it:

1. Add a urlTemplate-style EntryPoint object for the target
2. Validate the query-input names the same parameter as the placeholder
3. Support a second SearchAction for an app deep link
4. Change the placeholder name to `{search_term}` and update `query-input` to match.

⚠ Common Mistakes

- Forgetting the `{query}` placeholder in `target` means the engine has nowhere to inject the search term.
- Mismatching the placeholder name and the `name=` in `query-input` breaks the binding.
- Using `potentialActions` (plural) instead of the singular key fails the schema.

Skills gained SearchAction | potentialAction | string membership | WebSite schema

→ Where You'll Use This: Site-search-enabled homepages add this so Google can show a search box in the brand's result, letting users query the site straight from the SERP.

Mental Model

Like handing the search engine a fill-in-the-blank form: {query} is the blank, and the engine writes the visitor's words into it.

Project 331 Mark Text as Speakable

The Idea Behind It

The speakable property tells voice assistants which parts of a page to read aloud, chosen by CSS selectors. This code builds a WebPage whose speakable is a SpeakableSpecification listing a cssSelector array targeting the headline and summary. The code reads that selector list, reports its length, and joins the selectors into a readable line for inspection.

What You'll Build & Learn

Mark a WebPage's headline and summary as speakable via CSS selectors.

- speakable targets voice assistants
- SpeakableSpecification holds cssSelector
- Selectors choose readable regions

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 330: build a sitelinks searchbox. Project 332 changes the surface problem while preserving the underlying method.

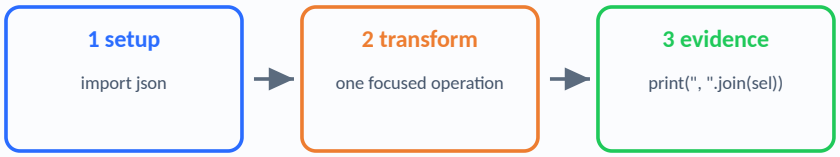
progress check

Reading target: mark text as speakable is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(", ".join(sel))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`"@context": "https://schema.org",`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(", ".join(sel))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `"@context": "https://schema.org",`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Mark Text as Speakable: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, page, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: "@context": "https://schema.org", | Evidence: print(", ".join(sel))

🛠️ Reader to Writer Bridge

1. Read: explain mark text as speakable in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 330.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 332 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(", ".join(sel))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of mark text as speakable and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`speakable` is a `SpeakableSpecification` whose `cssSelector` array points at the on-page elements an assistant should voice, joined here for a readable summary.

```
import json
page = {
    "@context": "https://schema.org",
    "@type": "WebPage",
    "name": "Daily News",
    "speakable": {
        "@type": "SpeakableSpecification",
        "cssSelector": [".headline", ".summary"],
    },
}
sel = page["speakable"]["cssSelector"]
print("speakable selectors:", len(sel))
print(", ".join(sel))
```

```
output
speakable selectors: 2
.headline, .summary
```

How it works:

1. The page dict declares WebPage with a name.
2. speakable is a SpeakableSpecification whose cssSelector lists .headline and .summary.
3. sel aliases that list, len reports 2 selectors, and join prints them comma-separated.

Try it:

1. Support xpath instead of cssSelector as an alternative
2. Validate each selector starts with a dot or hash
3. Add a third selector for the article lede
4. Add a `.byline` selector and watch the count rise.

⚠ Common Mistakes

- Putting a single selector as a bare string instead of a list makes `len()` count characters.
- Using an `xpath` array and a `cssSelector` array together when only one addressing scheme should be chosen.
- Selecting a hidden or boilerplate element that should not be read aloud.

Skills gained `speakable` `SpeakableSpecification` `cssSelector` `str.join`

→ Where You'll Use This: News and article pages mark their key passages speakable so voice assistants can read the headline and summary aloud in audio search and smart-speaker results.

⚡ Mental Model

Like highlighting the lines an audiobook narrator should read; the CSS selectors are the highlighter marking what gets voiced.

Project 332 Validate Price Is a Numeric String

💡 The Idea Behind It

An Offer price must be a bare number because priceCurrency carries the unit separately. This code loops over offers, pulls each price, and tries float on it inside a try block. A clean parse prints valid; a value with a currency symbol or the word free raises ValueError and prints not numeric, catching the two common mistakes that break the price.

What You'll Build & Learn

Validate that each Offer price is a parseable bare number.

- Price must be a bare numeric string
- priceCurrency carries the unit

- float parsing flags bad prices

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 331: mark text as speakable. Project 333 changes the surface problem while preserving the underlying method.

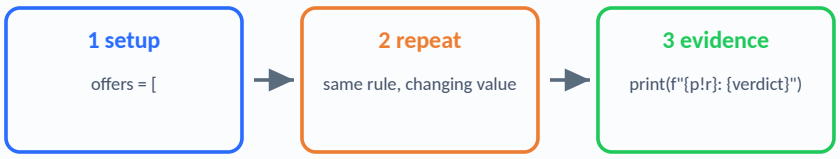
progress check

Reading target: validate price is a numeric string is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `offers = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"{p!r}: {verdict}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`offers = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`for o in offers:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f"{p!r}: {verdict}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for o in offers:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Read the try path as the hoped-for route and the catch/except/finally path as the boundary plan.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate Price Is a Numeric String: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: offers, price, priceCurrency, free, verdict. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: offers = [| Core: for o in offers: | Evidence: print(f"{p!r}: {verdict}")

Reader to Writer Bridge

1. Read: explain validate price is a numeric string in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 331.
   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 333 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{p!r}: {verdict}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `offers = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate price is a numeric string and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`float()` succeeds only on a clean numeric string, so wrapping it in a try/except cleanly separates schema-legal prices from ones polluted by symbols or words.

```
offers = [
    {"price": "19.99", "priceCurrency": "USD"},
    {"price": "$19.99", "priceCurrency": "USD"},
    {"price": "free", "priceCurrency": "USD"},
]
for o in offers:
    p = o["price"]
    try:
        float(p)
        verdict = "valid"
    except ValueError:
        verdict = "not numeric"
    print(f"{p!r}: {verdict}")
```

```
output
'19.99': valid
'$19.99': not numeric
'free': not numeric
```

How it works:

1. offers lists prices 19.99, $19.99, and free.
2. The loop reads each price into p.
3. float(p) inside a try sets valid; the except ValueError sets not numeric, so the symbol and the word both fail while the bare number passes.

Try it:

1. Strip a leading currency symbol then re-validate
2. Reject negative prices even though they parse
3. Report the offending offers as a list at the end
4. Add a thousands-separated price like `1,299.00` and see it rejected.

⚠ Common Mistakes

- Embedding a currency symbol in `price` (`$19.99`) duplicates what `priceCurrency` already states and fails validation.
- Using a comma decimal separator (`19,99`) also fails `float()` and needs locale handling.
- Catching every exception instead of `ValueError` would hide real bugs.

Skills gained float parsing | try/except | Offer price | ValueError

→ Where You'll Use This: Feeds often arrive with currency symbols glued to the price; this validation forces the markup to use a clean numeric price with the currency declared separately as Google requires.

Mental Model

Like a scale that weighs only the goods, not the packaging; the price field holds the number alone and the currency rides separately.

Project 333 Validate a Currency Code Is ISO 4217

The Idea Behind It

The priceCurrency must be a three-letter ISO 4217 code in uppercase, such as USD or EUR. This code holds a set of valid codes, then loops over offers checking each currency for membership. A correct code passes while a symbol like US$ or a lowercase eur fails, catching both symbol intrusion and case drift in one membership test.

What You'll Build & Learn

Validate each priceCurrency against a set of ISO 4217 codes.

- priceCurrency is an ISO 4217 code
- It must be uppercase three letters
- Set membership rejects symbols and case drift

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 332: validate price is a numeric string. Project 334 changes the surface problem while preserving the underlying method.

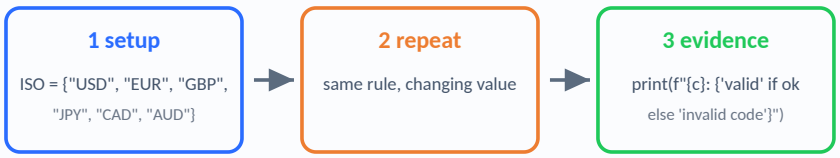
progress check

Reading target: validate a currency code is iso 4217 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `ISO = {"USD", "EUR", "GBP", "JPY", "CAD", "AUD"}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. repeat, test, update
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"{c}: {'valid' if ok else 'invalid code'}")` exposes the result; match it against the output before changing the program. output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`ISO = {"USD", "EUR", "GBP", "JPY", "CAD", "AUD"}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"{c}: {'valid' if ok else 'invalid code'}")`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f"{c}: {'valid' if ok else 'invalid code'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `print(f"{c}: {'valid' if ok else 'invalid code'}")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate a Currency Code Is ISO 4217: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: offers, priceCurrency, US, eur, ok. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: ISO = {"USD", "EUR", "GBP", "JPY", "CAD", "AUD"} | Core: print(f"{c}: {'valid' if ok else 'invalid code'}") | Evidence: print(f"{c}: {'valid' if ok else 'invalid code'}")

🛠️ Reader to Writer Bridge

1. Read: explain validate a currency code is iso 4217 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 332.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 334 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{c}: {'valid' if ok else 'invalid code'}")`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `ISO = {"USD", "EUR", "GBP", "JPY", "CAD", "AUD"}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate a currency code is iso 4217 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A set-membership test enforces the closed ISO 4217 vocabulary, catching both the symbol `US$` and the lowercase `eur` that schema.org would not accept.

```
ISO = {"USD", "EUR", "GBP", "JPY", "CAD", "AUD"}
offers = [
    {"priceCurrency": "USD"},
    {"priceCurrency": "US$"},
    {"priceCurrency": "eur"},
]
for o in offers:
    c = o["priceCurrency"]
    ok = c in ISO
    print(f"{c}: {'valid' if ok else 'invalid code'}")
```

```
output
USD: valid
US$: invalid code
eur: invalid code
```

How it works:

1. ISO is a set of six valid uppercase currency codes.
2. offers lists USD, the symbol US$, and lowercase eur.
3. The loop reads each c, tests c in ISO, and prints valid or invalid code, rejecting both the symbol and the lowercase value.

Try it:

1. Uppercase each code before checking to tolerate case drift
2. Load the full ISO 4217 list rather than six codes
3. Map a symbol like $ to USD before validating
4. Uppercase the value first so `eur` passes.

⚠ Common Mistakes

- Allowing lowercase by not normalising means `eur` slips through if you `.upper()` before checking but forget the codes are case-sensitive in the data.
- Confusing the currency symbol (`$`) with the ISO code (`USD`).

- Maintaining only a partial ISO set rejects valid but uncommon codes.

Skills gained set membership ISO 4217 currency validation loop

→ Where You'll Use This: Catalog feeds sometimes carry currency symbols or lowercase codes; this gate enforces the exact uppercase three-letter codes the structured-data spec demands before the price ships.

Mental Model

Like an international form that only accepts the airport-style three-letter code; a dollar sign or lowercase entry is simply not recognized.

Project 334 Validate an ISO 8601 Duration

The Idea Behind It

Schema durations such as cookTime use ISO-8601's PnYnMnDTnHnMnS form, most often the simple PT15M. This code compiles a regex anchoring the PT prefix with optional hour, minute, and second groups, then matches each candidate. A guard rejects a bare PT with no components. Valid durations pass while a phrase like 15 min fails, enforcing the format.

What You'll Build & Learn

Validate duration strings against the ISO-8601 PT pattern with a regex.

- Durations use the PT prefix form
- A regex anchors optional H, M, S groups
- Guard against an empty PT

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 333: validate a currency code is iso 4217. Project 335 changes the surface problem while preserving the underlying method.

progress check

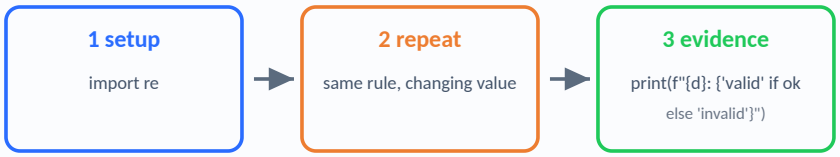
Reading target: validate an iso 8601 duration is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"{d}: {'valid' if ok else 'invalid'}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"{d}: {'valid' if ok else 'invalid'}")`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{d}: {'valid' if ok else 'invalid'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"{d}: {'valid' if ok else 'invalid'}")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate an ISO 8601 Duration: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: re, durations, min, pattern, compile. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: print(f"{d}: {'valid' if ok else 'invalid'}") | Evidence: print(f"{d}: {'valid' if ok else 'invalid'}")

🛠️ Reader to Writer Bridge

1. Read: explain validate an iso 8601 duration in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 333.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 335 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{d}: {'valid' if ok else 'invalid'}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of validate an iso 8601 duration and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The pattern anchors on `PT` and accepts optional hour/minute/second groups; the extra `d != 'PT'` guard rejects an empty duration that the optional groups would otherwise allow.

```
import re
durations = ["PT15M", "PT1H30M", "15 min", "PT0S"]
pattern = re.compile(r"^PT(?:\d+H)?(?:\d+M)?(?:\d+S)?$")
for d in durations:
    ok = bool(pattern.match(d)) and d != "PT"
    print(f"{d}: {'valid' if ok else 'invalid'}")
```

```
output
PT15M: valid
PT1H30M: valid
15 min: invalid
PT0S: valid
```

How it works:

1. durations lists PT15M, PT1H30M, the phrase 15 min, and PT0S.
2. pattern compiles a regex requiring PT then optional digit-H, digit-M, digit-S groups.
3. ok combines a successful match with a check that d is not bare PT, so 15 min fails and the rest pass.

Try it:

1. Extend the pattern to allow the date portion like P1DT2H
2. Capture the components and convert to total seconds
3. Reject PT0S as a zero-length duration
4. Add a day component by extending the pattern with `(?:\\d+D)?` after `PT` handling.

⚠ Common Mistakes

- Without the `d != 'PT'` guard the bare `PT` (no components) matches and is wrongly accepted.
- Forgetting to anchor with `^` and `$` lets trailing junk like `PT15Mxx` match.
- Writing a human duration `15 min` instead of the ISO form is the exact error this catches.

Skills gained re.compile | regex anchors | ISO-8601 duration | boolean guard

→ Where You'll Use This: cookTime, duration, and similar fields must be ISO durations; this regex validation catches a human writing 15 min or 1h30 instead of the machine-readable PT form.

⚡ Mental Model

Like a strict timecode format: the engine reads PT15M instantly but stares blankly at the human phrase fifteen minutes.

Project 335 Parse JSON-LD Back From a String

💡 The Idea Behind It

Auditing real pages means reading JSON-LD text back into Python. This code holds a raw JSON string, calls json.loads to turn it into a dict, then inspects the result by printing the @type, the headline, and the sorted keys. The round-trip from text to dict is what lets you examine markup pulled from a live page field by field.

What You'll Build & Learn

Parse a JSON-LD string into a dict and inspect its fields.

- json.loads parses text into a dict
- Inspect parsed fields by key
- sorted lists the keys

🧭 Skill Roadmap

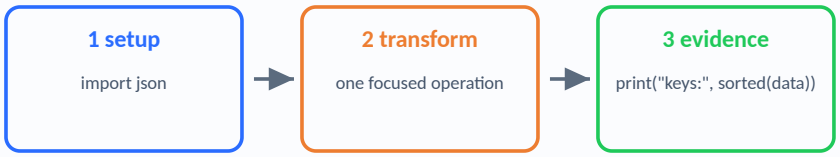
Skill step: pushes the same idea into a less automatic situation. It builds on Project 334: validate an iso 8601 duration. Project 336 changes the surface problem while preserving the underlying method. *progress check*

Reading target: parse json-ld back from a string is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("keys:", sorted(data))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`raw = '{"@context": "https://schema.org", "@type": "Article", "headline": "Hi"}'`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. slow line
3. Check the evidence	`print("keys:", sorted(data))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `raw = '{"@context": "https://schema.org", "@type": "Article", "headline": "Hi"}'`

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Parse JSON-LD Back From a String: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, raw, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: raw = '{"@context": "https://schema.org", "@type": "Article", "headline... | Evidence: print("keys:", sorted(data))

🛠️ Reader to Writer Bridge

1. Read: explain parse json-ld back from a string in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 334.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 336 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("keys:", sorted(data))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of parse json-ld back from a string and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`json.loads` reverses `json.dumps`, parsing the block's text into a dict whose keys (sorted here for stable output) can then be audited.

```
import json
raw = '{"@context": "https://schema.org", "@type": "Article", "headline":
"Hi"}'
data = json.loads(raw)
print("type:", data["@type"])
```

```
print("headline:", data["headline"])
print("keys:", sorted(data))
```

```
output
type: Article
headline: Hi
keys: ['@context', '@type', 'headline']
```

How it works:

1. raw is a JSON string declaring an Article with a headline.
2. json.loads(raw) parses it into the data dict.
3. The prints read data['@type'] and data['headline'], and sorted(data) lists the three keys alphabetically.

Try it:

1. Catch a json.JSONDecodeError on malformed input
2. Parse a string holding an @graph array
3. Pretty-print the parsed dict with json.dumps and indent
4. Add a nested `author` object to the raw string and read it.

⚠ Common Mistakes

- Single-quoting the JSON inside the string makes it invalid JSON; JSON requires double quotes.
- A trailing comma in the source string raises `json.JSONDecodeError`.
- Assuming `data` is a list when a single block parses to a dict (or vice versa for an array).

Skills gained json.loads | parsing | dict inspection | sorted

→ Where You'll Use This: Every page-auditing pipeline starts by parsing the JSON-LD text from a crawl into Python objects so the rest of your validation logic can examine the structured data.

Mental Model

Like decoding a sealed message back into plain words: json.loads opens the envelope so you can read each line.

Project 336 Extract JSON-LD From a Script Tag

The Idea Behind It

On a live page the JSON-LD sits inside a script tag of type application/ld+json. This code holds an HTML string, uses a regex to capture the inner text between the opening and closing script tags, then feeds that captured

group to json.loads. The escaped plus sign in the pattern matches the literal ld+json, and the resulting dict is inspected by key.

What You'll Build & Learn

Extract JSON-LD from an HTML script tag and parse it.

- JSON-LD lives in a typed script tag
- Regex captures the inner text
- Escape the plus in ld+json

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 335: parse json-ld back from a string. Project 337 changes the surface problem while preserving the underlying method.

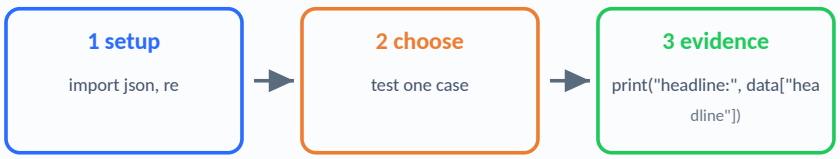
progress check

Reading target: extract json-ld from a script tag is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json, re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("headline:", data["headline"])` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json, re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`'<head><title>Page</title>'`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print("headline:", data["headline"])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `'<head><title>Page</title>'`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Extract JSON-LD From a Script Tag: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, re, html, head, title. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json, re | Core: '<head><title>Page</title>' | Evidence: print("headline:", data["headline"])

Reader to Writer Bridge

1. Read: explain extract json-ld from a script tag in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 335.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 337 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("headline:", data["headline"])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json, re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of extract json-ld from a script tag and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A non-greedy `(.*?)` capture grabs only the script tag's inner JSON, which `json.loads` parses; the `\+` escapes the literal plus in `ld+json`.

```
import json, re
html = (
    '<head><title>Page</title>'
    '<script type="application/ld+json">'
    '{"@type": "Article", "headline": "Found Me"}'
    '</script></head>'
)
m = re.search(r'<script type="application/ld\+json">(.*?)</script>', html)
data = json.loads(m.group(1))
print("extracted @type:", data["@type"])
print("headline:", data["headline"])
```

```
output
extracted @type: Article
headline: Found Me
```

How it works:

1. html is a string containing a head with a script of type application/ld+json holding an Article.
2. re.search captures the inner JSON with a pattern escaping the plus.
3. json.loads(m.group(1)) parses it, and the prints show the @type and headline.

Try it:

1. Use re.findall to extract multiple script blocks from one page
2. Add re.DOTALL so the pattern spans newlines
3. Fall back gracefully when no script tag matches
4. Add a second JSON-LD script and switch to `re.findall` to get both.

⚠ Common Mistakes

- Forgetting to escape the `+` in `ld+json` makes the regex match a repeated `d`, not the literal tag.
- A greedy `(.*)` would swallow everything up to the last `</script>` on a multi-block page.
- Regex-parsing HTML is brittle; a real auditor would use an HTML parser for nested or attribute-reordered tags.

Skills gained re.search | regex capture | json.loads | HTML extraction

→ Where You'll Use This: Scraping structured data from fetched pages means locating each ld+json script tag and parsing its contents, the core step in any large-scale structured-data audit crawler.

⚡ Mental Model

Like fishing a note out of a labeled drawer: the script tag is the drawer, the regex is the hand that pulls the note, json.loads reads it.

Project 337 Validate AggregateOffer Low and High

💡 The Idea Behind It

An AggregateOffer summarizes many sellers with a lowPrice and a highPrice, and the low must not exceed the high. This code reads both prices, floats them to compare numerically, and checks lo <= hi. A valid band prints valid range; an inverted pair would print low > high. It is a quick numeric sanity check on the price span.

What You'll Build & Learn

Validate that an AggregateOffer's low price does not exceed its high.

- AggregateOffer spans many sellers
- lowPrice must not exceed highPrice
- float for numeric comparison

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 336: extract json-ld from a script tag. Project 338 changes the surface problem while preserving the underlying method. *progress check*

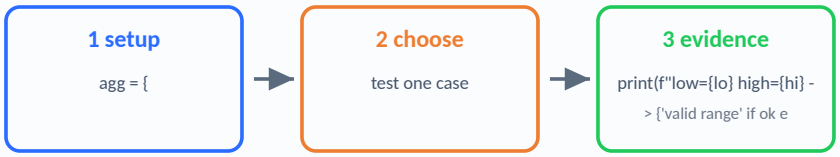
Reading target: validate aggregateoffer low and high is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `agg = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}")` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`agg = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}")`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}")`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Validate AggregateOffer Low and High: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: agg, type, AggregateOffer, lowPrice, highPrice. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: agg = { | Core: print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}") | Evidence: print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}")

Reader to Writer Bridge

1. Read: explain validate aggregateoffer low and high in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 336.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 338 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `agg = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate aggregateoffer low and high and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Casting the price strings to floats lets a numeric comparison confirm the band is ordered; an AggregateOffer with low above high is incoherent.

```
agg = {
    "@type": "AggregateOffer",
    "lowPrice": "29.00",
    "highPrice": "59.00",
    "priceCurrency": "USD",
    "offerCount": 4,
}
lo = float(agg["lowPrice"])
hi = float(agg["highPrice"])
ok = lo <= hi
print(f"low={lo} high={hi} -> {'valid range' if ok else 'low > high'}")
```

```
output
low=29.0 high=59.0 -> valid range
```

How it works:

1. The agg dict declares AggregateOffer with lowPrice, highPrice, priceCurrency, and offerCount.
2. lo and hi float-parse the two price strings.
3. ok checks lo <= hi, and the f-string prints the numeric low and high plus a valid range or low > high verdict.

Try it:

1. Also confirm offerCount is a positive integer
2. Validate both prices parse as numbers before comparing
3. Check the currency code against an ISO set as well
4. Swap the two prices and watch the verdict flip.

⚠ Common Mistakes

- Comparing the price strings directly (`'29.00' <= '59.00'`) happens to work here but fails lexically once values differ in length, e.g. `'100' < '29'`.
- Forgetting `offerCount` leaves the aggregate without the seller tally it should report.
- Swapping `lowPrice` and `highPrice` produces a silently inverted, invalid band.

Skills gained float | AggregateOffer | range check | f-strings

→ Where You'll Use This: Comparison and marketplace pages emit an AggregateOffer for a product sold by many merchants; this guards against a feed bug that swaps the low and high price bounds.

Mental Model

Like a price range on a listing: the from price must sit below the to price, or the band reads backwards and makes no sense.

Project 338 Audit a Full Product JSON-LD Block

The Idea Behind It

This capstone ties the chapter together with one audit over a Product block. It builds a checks dict whose values are boolean tests of the context, type, name, image, and the nested offer price, using get to read fields safely. Looping prints PASS or FAIL per check, and all over the values declares overall rich-result eligibility from the combined verdicts.

What You'll Build & Learn

Run a combined audit of a Product block and declare eligibility.

- A checks dict maps names to boolean tests
- dict.get reads fields safely
- all over the values gives the verdict

Skill Roadmap

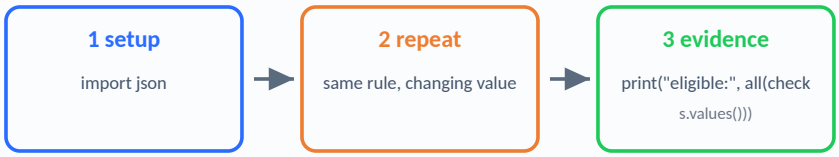
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 337: validate aggregateoffer low and high. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: audit a full product json-ld block is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
- Evidence: `print("eligible:", all(checks.values()))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
2. Apply the rule	`print(f"{k}: {'PASS' if ok else 'FAIL'}")`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print("eligible:", all(checks.values()))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Structured Data and Rich Results becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f"{k}: {'PASS' if ok else 'FAIL'}")`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Audit a Full Product JSON-LD Block: connect the surface task to the chapter skill, generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.
Names	Watch these names first: json, product, context, https, schema. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: print(f"{k}: {'PASS' if ok else 'FAIL'}") | Evidence: print("eligible:", all(checks.values()))

🛠️ Reader to Writer Bridge

1. Read: explain audit a full product json-ld block in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 337.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print("eligible:", all(checks.values()))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of audit a full product json-ld block and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Each named check evaluates to a boolean, and `all(checks.values())` collapses them into a single eligibility verdict, the skeleton of a real structured-data linter.

```
import json
product = {
    "@context": "https://schema.org",
    "@type": "Product",
    "name": "Trail Runner Shoe",
    "image": "https://example.com/shoe.jpg",
    "offers": {
        "@type": "Offer",
        "price": "89.99",
        "priceCurrency": "USD",
        "availability": "https://schema.org/InStock",
    },
}
checks = {
    "@context": product.get("@context") == "https://schema.org",
    "@type": product.get("@type") == "Product",
    "name": bool(product.get("name")),
    "image": bool(product.get("image")),
    "offer price": "price" in product.get("offers", {}),
}
for k, ok in checks.items():
    print(f"{k}: {'PASS' if ok else 'FAIL'}")
print("eligible:", all(checks.values()))
```

```
output
@context: PASS
@type: PASS
name: PASS
image: PASS
offer price: PASS
eligible: True
```

How it works:

1. product is a full Product with name, image, and a nested Offer.
2. The checks dict tests @context, @type, name, image, and offer price using get and bool.
3. The loop prints PASS or FAIL per check, and all(checks.values()) prints eligible True.

Try it:

1. Add a check that priceCurrency is a valid ISO code
2. Report only the failing check names at the end
3. Generalize the audit to dispatch checks by the block's @type
4. Remove the `image` key and watch eligibility turn False.

⚠ Common Mistakes

- Using `product['offers']['price']` instead of `'price' in product.get('offers', {})` raises KeyError when `offers` is absent.
- Checking `product['name']` truthiness with `bool(...)` treats an empty string as a fail, which is the intent but surprises if names can legitimately be blank.
- Returning eligibility before printing the per-field results hides why a block failed.

Skills gained dict.get | all | boolean checks | Product audit

→ Where You'll Use This: This is the template for a real per-page structured-data linter: a named set of boolean assertions whose combined result is a single ship-or-fix verdict for each block.

⚡ Mental Model

Like a pre-flight checklist: each item gets a green or red light, and the plane only takes off when every light is green.

✅ Chapter Progress Checkpoint

What this chapter added: after 37 mini-projects, Structured Data and Rich Results should feel like a reusable Python move: generate and validate json-ld for article, product, faq, localbusiness and breadcrumb.

chapter checkpoint

1 Read
Open any project from Projects 302-338 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 10 (International SEO and hreflang) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

8 questions · 4 categories · graded

2 Recall | 2 Meaning | 3 Structure | 1 Transfer

9.1.Q1 Recall MEDIUM Turn back to Project 307 (Count Question and Answer Pairs). Without running it, what exactly does it print?

9.1.Q2 Meaning MEDIUM Turn back to Project 309 (Validate Breadcrumb Position Ordering). Without running it, what exactly does it print?

9.1.Q3 Structure MEDIUM Turn back to Project 322 (Check Rich-Result Eligibility). Without running it, what exactly does it print?

9.1.Q4 Transfer MEDIUM Turn back to Project 336 (Extract JSON-LD From a Script Tag). Without running it, what exactly does it print?

9.1.Q5 Recall EASY In a JSON-LD block, what do the @context and @type keys do?

A @context sets the page URL; @type sets the title

B @context names the vocabulary (schema.org); @type names what the object is

C They are optional comments ignored by engines

D @context is the author; @type is the date

9.1.Q6 Meaning MEDIUM A Product block has a name and an image but no offers, while the required set for Product is name and offers. What is the result?

A It still gets a full shopping rich result

B It is not eligible for the rich result until offers is added

C The engine invents a default price

D It is rejected as spam

9.1.Q7 Structure EASY Why is a misspelled @type value like "Prodcut" especially dangerous?

A It crashes the page

B It raises a visible validation error you cannot miss

C The engine silently ignores the unknown type, so the block does nothing and you may not notice

D It changes the page URL

9.1.Q8 Structure MEDIUM A Product's offers value needs a price, currency and stock state. How is that expressed in JSON-LD, and why?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 9 – Structured Data and Rich Results

9.1.Q1. `questions=3 answered=3`

9.1.Q2. `positions=[1, 2, 3] -> ordered`

9.1.Q3. `A -> eligible` then `B -> needs ['image']`

9.1.Q4. `extracted @type: Article` then `headline: Found Me`

9.1.Q5. B @context says which vocabulary the field names come from (always "https://schema.org" for SEO), and @type says what kind of thing the object describes.

9.1.Q6. B a type is eligible only when every required property is present; missing the required offers means no rich result until it is added.

9.1.Q7. C unknown types are silently dropped rather than flagged, so the structured data quietly stops working.

9.1.Q8. As a nested Offer object (its own @type with price, priceCurrency and availability) — because a value with internal structure becomes a typed sub-dictionary rather than a plain string.



PART III

Advanced Mastery

Push into the professional toolkit: concurrency, design patterns, performance, data work and the algorithms behind real systems.

Chapter 10 – International SEO and hreflang · Chapter 11 – Off-Page, Links and Authority · Chapter 12 – Local and Map SEO · Chapter 13 – Log-File and Server Analysis · Chapter 14 – GEO, AEO and the End-to-End Technical Audit

Chapter 10 · International SEO and hreflang

Validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

Chapter Skill Map

What this chapter adds: In International SEO and hreflang, the focus is validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay. You practise it across 32 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples.

chapter arc

What it builds on: Chapter 10 assumes the reading habits from Chapter 9: Structured Data and Rich Results. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 11, Off-Page, Links and Authority, reuses this chapter's discipline with a new kind of problem.

1 **Projects 339-348**	2 **Projects 349-358**	3 **Projects 359-370**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

hreflang is a bidirectional map of locale → URL that says *"for this audience, serve this page."*

same content, different audience

When the same content exists in several languages or for several countries, a search engine faces a choice it cannot make from the words alone: which version should it show a French user in Canada, or an English user in Britain? The hreflang annotation answers that. On each page you list every alternate version as a *(code, URL)* pair — `hreflang="fr-CA"` points at the Canadian-French URL, `hreflang="en-GB"` at the British-English one. The engine then serves the version whose code best matches the user's language and region.

The code itself has a strict shape: a language subtag, optionally followed by a region subtag. Language comes from ISO 639-1 (`en`, `fr`, `de`) and is written lowercase; region comes from ISO 3166-1 alpha-2 (`US`, `GB`, `CA`) and is written uppercase. So `en` means "English anywhere" and `en-US` means "English, United States." A single regex captures the legal form before you trust any tag:

```
import re
codes = ['en', 'en-US', 'fr-CA', 'English', 'en_US', 'de-de']
pat = re.compile(r'^[a-z]{2}(-[A-Z]{2})?$')
for c in codes:
    print(f'{c}: {"valid" if pat.match(c) else "invalid"}')
```

```
output
```

```
en: valid
en-US: valid
fr-CA: valid
English: invalid
en_US: invalid
de-de: invalid
```

Notice what the regex rejects: `English` (not a two-letter code), `en_US` (underscore, not hyphen), and `de-de` (lowercase region). A crawler silently *skips* a malformed value, so the tag you carefully wrote simply does nothing — the most expensive kind of bug, because it looks like it is working.

The rule that catches most teams is `reciprocity`: hreflang is a two-way handshake. If page A says "fr is over there at B," then B must say "en is back here at A." A link without a return tag is ignored by Google. So an hreflang cluster is not a list you write once on the homepage — it is the *same block, repeated on every page in the set*, each page including itself.

a one-way hreflang is no hreflang

Finally, hreflang lives alongside the `canonical` tag, and the two must agree. hreflang says "these pages are equals for different audiences"; canonical says "this page is the indexable original." If your French page canonicalizes to the English one, you have told Google to *drop French from the index* — and your hreflang for French now points at a page that will never appear. The safe rule: every localized page canonicalizes to *itself*.

HREFLANG

An annotation (in the page `<head>`, an HTTP header, or the XML sitemap) listing each alternate language/region version of a page as a `(code, URL)` pair, so the engine can serve the right version per user.

LOCALE CODE

The value of an hreflang attribute: a language subtag, optionally a hyphen and a region subtag — `en`, `en-US`, `fr-CA`. Plus the special value `x-default`.

ISO 639-1

The standard for two-letter language codes — `en`, `fr`, `de`, `ja`. The language subtag of an hreflang code must be a real 639-1 code, written lowercase (`jp` is not the code for Japanese; `ja` is).

ISO 3166-1 ALPHA-2

The standard for two-letter country/region codes — US, GB, CA, BR. The optional region subtag must be one of these, written uppercase. Three-letter or numeric codes are not allowed in hreflang.

RETURN TAG (RECIPROCITY)

For hreflang to count, if page A links to page B as an alternate, B must link back to A. A missing return tag means the engine ignores the relationship entirely.

X-DEFAULT

The special hreflang value naming the page to serve users whose language/region matches none of your locales — typically a language selector or neutral root page.

SELF-REFERENCE

The rule that every page must list its *own* locale among its hreflang alternates. A page missing its self-reference invalidates the whole cluster for that page.

Read the code: language is not region

Every hreflang value decomposes into at most two parts split by a hyphen. The first is the `language` (ISO 639-1, lowercase); the second, if present, is the `region` (ISO 3166-1 alpha-2, uppercase). Two letters alone is not enough — the language must be a *real* 639-1 code. The classic trap is `jp` for Japanese; the correct code is `ja`. Validate the language subtag against a known set, not just its length:

```
known = {'en', 'fr', 'de', 'es', 'ja', 'zh'}
codes = ['en-US', 'fr', 'xx-GB', 'de-DE', 'jp-JP']
for c in codes:
    lang = c.split('-')[0]
    ok = lang in known
    print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')
```

```
output
en-US: lang=en known
fr: lang=fr known
xx-GB: lang=xx UNKNOWN
de-DE: lang=de known
jp-JP: lang=jp UNKNOWN
```

The region subtag is just as strict: exactly two letters, uppercase, and a real 3166-1 code. That rules out three-letter codes like `DEU`, lowercase `us`, and the numeric UN region codes such as `419` (Latin America) that some standards allow but hreflang does not. A casing swap — uppercase language, lowercase region, as in `EN-us` — is the single most common typo, and it is silently dropped:

```
codes = ['en-US', 'en-us', 'fr-CA', 'de-DEU', 'es-419', 'pt-BR']
for c in codes:
    if '-' not in c:
        print(f'{c}: no region'); continue
    region = c.split('-', 1)[1]
    ok = len(region) == 2 and region.isalpha() and region.isupper()
    print(f'{c}: region={region} {"valid" if ok else "INVALID"}')
```

```
output
en-US: region=US valid
en-us: region=us INVALID
fr-CA: region=CA valid
de-DEU: region=DEU INVALID
es-419: region=419 INVALID
pt-BR: region=BR valid
```

The cluster and the reciprocity handshake

The set of all language alternates for one piece of content is its hreflang cluster. The defining property of a correct cluster: the hreflang block is identical on every page in it — each page lists all locales, including itself. That symmetry is what gives you reciprocity for free. Generate the block once from the cluster and stamp it on every URL, and you can never produce a one-way link:

```
cluster = {'en': 'https://s.com/en/', 'fr': 'https://s.com/fr/',
           'de': 'https://s.com/de/'}
for page_code in sorted(cluster):
    print(f'On {cluster[page_code]}:')
    for code in sorted(cluster):
        print(f'  hreflang="{code}" -> {cluster[code]}')
```

```
output
On https://s.com/de/:
  hreflang="de" -> https://s.com/de/
  hreflang="en" -> https://s.com/en/
  hreflang="fr" -> https://s.com/fr/
On https://s.com/en/:
  hreflang="de" -> https://s.com/de/
  hreflang="en" -> https://s.com/en/
  hreflang="fr" -> https://s.com/fr/
On https://s.com/fr/:
  hreflang="de" -> https://s.com/de/
  hreflang="en" -> https://s.com/en/
  hreflang="fr" -> https://s.com/fr/
```

When clusters are edited by hand they drift, and a return tag goes missing. The audit is direct: for every alternate a page declares, look up the *target's* hreflang block and confirm the original page is named back. Report the exact

missing edge — the team needs the URL to fix, not a bare "broken":

```
pages = {
    'https://s.com/en/': {'en': 'https://s.com/en/', 'es': 'https://s.com/es/'},
    'https://s.com/es/': {'es': 'https://s.com/es/'},
}
missing = []
for page, alts in pages.items():
    for target in alts.values():
        if page not in pages.get(target, {}).values():
            missing.append((page, target))
for src, tgt in missing:
    print(f'missing return: {tgt} should link back to {src}')
print('broken links:', len(missing))
```

```
output
missing return: https://s.com/es/ should link back to https://s.com/en/
broken links: 1
```

x-default and the self-reference

Two values must appear that are easy to forget. The first is `x-default` — the page to serve a user whose language matches none of your locales (a German speaker landing on an English/French/Spanish store). It usually points at a neutral language selector or the root, *distinct* from any specific locale URL. The second is the `self-reference`: every page must list its own locale among its alternates, or Google invalidates the whole set for that page. A quick cluster audit checks both at once:

```
cluster_codes = {'en-US', 'en-GB', 'fr-FR'}
page_locale = 'en-US'
checks = {
    'self present': page_locale in cluster_codes,
    'x-default present': 'x-default' in cluster_codes,
    'at least 2 locales': len(cluster_codes) >= 2,
}
for name in sorted(checks):
    print(f'{name}: {"PASS" if checks[name] else "FAIL"}')
```

```
output
at least 2 locales: PASS
self present: PASS
x-default present: FAIL
```

The `x-default` is also where the locale fallback chain ends. When a requested locale is not served, the engine falls back to the bare language, then to `x-default`. Walking that chain is how a server (or browser) negotiates which version a visitor actually gets:

```
available = {'en', 'en-GB', 'fr'}
def resolve(req):
    chain = [req, req.split('-')[0], 'x-default']
    for cand in chain:
        if cand in available:
            return cand
    return 'none'
for req in ['en-GB', 'en-US', 'fr-CA', 'de-DE']:
    print(f'{req} -> {resolve(req)}')
```

```
output
en-GB -> en-GB
en-US -> en
fr-CA -> fr
de-DE -> none
```

How hreflang and canonical interact

hreflang and the `canonical` tag answer different questions, and they must not contradict each other. canonical says *"this is the indexable version of this page"*; hreflang says *"these are equal alternates for other audiences."* The safe configuration is that `every localized page is self-canonical` — it points its canonical at itself, keeping each language version eligible for the index. Verify it directly:

```
pages = {
    'https://s.com/en/': 'https://s.com/en/',
    'https://s.com/fr/': 'https://s.com/fr/',
    'https://s.com/de/': 'https://s.com/en/',
}
for page in sorted(pages):
    canon = pages[page]
    print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon}')
```

```
output
https://s.com/de/: POINTS AWAY -> https://s.com/en/
https://s.com/en/: self
https://s.com/fr/: self
```

The damage from a `cross-locale canonical` is subtle but total: if the French page canonicalizes to the English page, you have told Google the two are the *same* page and English is the keeper — so French is dropped from the index, and your `fr` hreflang now points at a page that will never rank. The audit compares the *locale of the page* with the *locale of its canonical* and flags any mismatch:

```
locale_of = {'https://s.com/en/': 'en', 'https://s.com/fr/': 'fr'}
canonicals = {
    'https://s.com/en/': 'https://s.com/en/',
```

```
    'https://s.com/fr/': 'https://s.com/en/',
}
for page in sorted(canonicals):
    canon = canonicals[page]
    if locale_of[page] != locale_of.get(canon):
        print(f'{page} ({locale_of[page]}) canonicalises to {locale_of.get(canon)}
- WRONG')
    else:
        print(f'{page}: same-locale canonical OK')
```

```
output
https://s.com/en/: same-locale canonical OK
https://s.com/fr/ (fr) canonicalises to en - WRONG
```

One pass: the full audit

Put the pieces together and you have the skeleton of an international-SEO audit. Across a real *page → alternates* map, in a single sweep: validate every code's shape, confirm each alternate has a return tag, and check that `x-default` appears everywhere. That trio catches the great majority of hreflang failures in production:

```
import re
pages = {
    'https://s.com/en/': {'en': 'https://s.com/en/', 'fr': 'https://s.com/fr/', 'x-
default': 'https://s.com/'},
    'https://s.com/fr/': {'en': 'https://s.com/en/', 'fr': 'https://s.com/fr/', 'x-
default': 'https://s.com/'},
}
pat = re.compile(r'^[a-z]{2}(-[A-Z]{2})?$')
bad_codes, no_return = [], []
for page, alts in pages.items():
    for code, target in alts.items():
        if code == 'x-default':
            continue
        if not pat.match(code):
            bad_codes.append(code)
        if page not in pages.get(target, {}).values():
            no_return.append((page, target))
xd_ok = all('x-default' in a for a in pages.values())
print('pages audited:', len(pages))
print('invalid codes:', sorted(set(bad_codes)))
print('broken return tags:', len(no_return))
print('x-default everywhere:', xd_ok)
```

```
output
pages audited: 2
invalid codes: []
```

```
broken return tags: 0
x-default everywhere: True
```

Example 1

Are these four hreflang values legal? `en-US`, `EN-us`, `de-DEU`, `pt-BR`. For each invalid one, say exactly what is wrong.

SOLUTION

1. `en-US`: language `en` is lowercase 639-1, region `US` is uppercase two-letter 3166-1 — valid.
2. `EN-us`: the casing is swapped — language should be lowercase, region should be uppercase. Invalid.
3. `de-DEU`: the region `DEU` is three letters; hreflang requires the two-letter alpha-2 form `DE`. Invalid.
4. `pt-BR`: `pt` is a real language code and `BR` a valid region — valid.

Valid: en-US, pt-BR. Invalid: EN-us (casing swapped), de-DEU (three-letter region).

shape is lowercase-lang, optional uppercase-2-letter-region

Example 2

A two-page set: the English page lists `en → /en/, es → /es/`; the Spanish page lists only `es → /es/`. Does reciprocity hold? What breaks?

SOLUTION

1. Check the English page: it names `/es/` as an alternate, so the Spanish page must name `/en/` back.
2. Look at the Spanish page's block: it lists only `es → /es/` (its self-reference is present), but there is no return tag pointing at `/en/`.
3. Because the link is one-way, Google ignores the English→Spanish relationship entirely; the missing edge is the Spanish page's `en → /en/`.

Reciprocity is broken: `/es/` must link back to `/en/`. Until the Spanish page adds `en → /en/`, the cluster is invalid.

hreflang is a two-way handshake

Example 3

You serve `en`, `en-GB`, and `fr`. A visitor requests `en-US`. Which version do they get, and why?

SOLUTION

1. Try the exact requested locale `en-US` against what is available — it is not served.
2. Fall back to the bare language: `en-US`.split → `en`, which is available. Match.
3. No need to reach `x-default`; the language-level fallback already resolved.

They get `en` (generic English), via the chain en-US → en. The fallback walks exact → language → x-default.

bare language is the safety net before x-default

⚡ Mental-Math Shortcut

Same block on every page, each naming itself, all self-canonical. A correct hreflang cluster is one *(code → URL)* table repeated identically on every page in the set: every page lists every locale including its own, an x-default is present, and each page canonicalizes to itself. If you generate the block from one source instead of editing per page, reciprocity, self-reference, and consistency all come for free.

⚠ Common Mistakes

- Swapping the case — `EN-us` instead of `en-US`. Language is lowercase (ISO 639-1), region is uppercase (ISO 3166-1 alpha-2); crawlers silently skip a wrongly-cased code.
- Using `jp` for Japanese (the code is `ja`), `gb` as a language, or a country code where a language belongs. Two valid-looking letters are not enough; the subtag must be the real standard code.
- Forgetting the return tag, so page A links to B but B never links back — the relationship is ignored. Edit clusters from one generated source rather than by hand to avoid this.
- Canonicalizing a localized page to a different locale (e.g. French → English). That tells Google to drop the French page from the index, and the `fr` hreflang then points at a page that will never appear. Keep every locale self-canonical.
- Omitting the self-reference or the `x-default`: a page that does not list its own locale invalidates the whole set for that page, and a cluster with no x-default leaves international fallback to chance.

This chapter contains 32 hands-on mini-projects (Projects 339–370). Type each one out, run it, and experiment.

Project 339 Validate an Hreflang Value

💡 The Idea Behind It

A valid hreflang value is either a bare two-letter language or a language-region pair joined by a hyphen. This project compiles one regular expression, anchored at both ends, that demands lowercase letters for the language and an optional uppercase two-letter region, then tests six sample codes so you can see exactly which shapes pass and which fail.

What You'll Build & Learn

Build a single regex that accepts legal hreflang shapes and rejects everything else, proving the pattern on mixed input.

- The canonical hreflang shape lang or lang-REGION
- Anchoring a regex with caret and dollar
- Why casing and separators matter to crawlers

🧭 Skill Roadmap

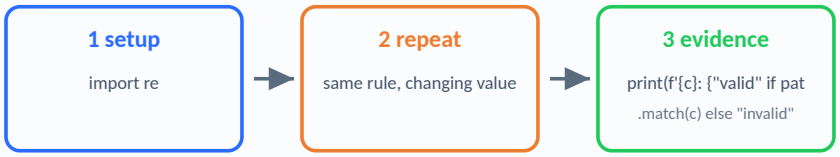
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 340 changes the surface problem while preserving the underlying method. *progress check*

Reading target: validate an hreflang value is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{c}: {"valid" if pat.match(c) else "invalid"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`print(f'{c}: {"valid" if pat.match(c) else "invalid"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{c}: {"valid" if pat.match(c) else "invalid"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{c}: {"valid" if pat.match(c) else "invalid"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate an Hreflang Value: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: re, codes, en, US, fr. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: print(f'{c}: {"valid" if pat.match(c) else "invalid"}') | Evidence: print(f'{c}: {"valid" if pat.match(c) else "invalid"}')

🛠️ Reader to Writer Bridge

1. Read: explain validate an hreflang value in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 340 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c}: {"valid" if pat.match(c) else "invalid"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate an hreflang value and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The pattern `^[a-z]{2}(-[A-Z]{2})?$` demands a two-letter lowercase language with an optional uppercase two-letter region, rejecting underscores, full words, and wrong casing.

```
import re
codes = ['en', 'en-US', 'fr-CA', 'English', 'en_US', 'de-de']
pat = re.compile(r'^[a-z]{2}(-[A-Z]{2})?$')
for c in codes:
    print(f'{c}: {"valid" if pat.match(c) else "invalid"}')
```

```
output
en: valid
en-US: valid
fr-CA: valid
English: invalid
en_US: invalid
de-de: invalid
```

How it works:

1. A list named codes holds six strings mixing valid and invalid forms like 'en', 'en-US', and 'English'.
2. re.compile builds the pattern '^[a-z]{2}(-[A-Z]{2})?$' once, with the region group made optional by the trailing question mark.
3. The loop calls pat.match on each code and prints 'valid' when it matches and 'invalid' otherwise, using an inline conditional inside an f-string.

Try it:

1. Extend the pattern to also accept the literal token 'x-default' as a special case.
2. Add codes with three-letter languages and confirm the regex rejects them.
3. Capture the language and region into named groups and print each part separately.
4. Add `'es-419'` and decide whether numeric UN M.49 regions should be allowed.

⚠ Common Mistakes

- Allowing `en_US` by matching `_` instead of `-` accepts a value Google ignores.
- Dropping the case anchors lets `EN-us` slip through even though hreflang is case-insensitive only on input, not on convention.
- Forgetting `^` and `$` lets the regex match a valid fragment inside a longer junk string.

Skills gained regex validation hreflang f-strings

→ **Where You'll Use This:** Every hreflang audit starts by separating well-formed codes from typos before any deeper checks; this regex is the gatekeeper you run first on scraped tag values.

Mental Model

The regex is a bouncer at the door: it does not care who you are, only that your ID has the right format before you get inside.

Project 340 Validate the ISO 639-1 Language Part

The Idea Behind It

Passing the shape test is not enough; the language subtag must also be a real ISO 639-1 code. Here a set of known languages is the source of truth, and each input is split on the hyphen to isolate its language part before membership is checked. Invented codes like 'xx' and the wrong code 'jp' for Japanese are caught even though they look structurally fine.

What You'll Build & Learn

Confirm the language portion of each code is a genuine ISO 639-1 entry by testing it against a known set.

- Shape validity differs from semantic validity
- Set membership for fast lookups
- Common wrong code jp versus ja

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 339: validate an hreflang value. Project 341 changes the surface problem while preserving the underlying method.

progress check

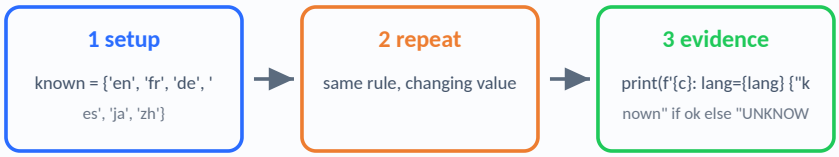
Reading target: validate the iso 639-1 language part is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `known = {'en', 'fr', 'de', 'es', 'ja', 'zh'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`known = {'en', 'fr', 'de', 'es', 'ja', 'zh'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate the ISO 639-1 Language Part: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: known, en, fr, de, es. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: known = {'en', 'fr', 'de', 'es', 'ja', 'zh'} | Core: print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}') | Evidence: print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')

🛠️ Reader to Writer Bridge

1. Read: explain validate the iso 639-1 language part in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 339.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 341 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `known = {'en', 'fr', 'de', 'es', 'ja', 'zh'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of validate the iso 639-1 language part and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Splitting on `-` isolates the language subtag, and membership in a curated ISO 639-1 set separates real languages from plausible-looking typos.

```
known = {'en', 'fr', 'de', 'es', 'ja', 'zh'}
codes = ['en-US', 'fr', 'xx-GB', 'de-DE', 'jp-JP']
for c in codes:
    lang = c.split('-')[0]
    ok = lang in known
    print(f'{c}: lang={lang} {"known" if ok else "UNKNOWN"}')
```

```
output
en-US: lang=en known
fr: lang=fr known
xx-GB: lang=xx UNKNOWN
de-DE: lang=de known
jp-JP: lang=jp UNKNOWN
```

How it works:

1. A set named known lists the real two-letter languages the site supports, such as 'en', 'fr', and 'ja'.
2. For each code in the list, split('-')[0] grabs the language part, dropping any region after the hyphen.
3. The boolean ok tests lang in known, and the f-string prints the extracted language plus 'known' or 'UNKNOWN'.

Try it:

1. Load the full ISO 639-1 list from a file instead of a small hard-coded set.
2. Print a suggestion when 'jp' is seen, pointing the editor to 'ja'.
3. Count how many codes fail and report the failure rate as a percentage.
4. Add `'pt'` to the known set and re-test a `pt-BR` code.

⚠ Common Mistakes

- Assuming any two lowercase letters is a valid language misses that `jp` is not the ISO code for Japanese (`ja` is).
- Splitting on `_` instead of `-` never isolates the language part of a hyphenated code.
- Treating the region (`US`) as the language by taking the second part inverts the check.

Skills gained sets iso-639-1 string split membership

→ Where You'll Use This: Auditors use a whitelist of supported languages so that a code which is the right shape but a fictional or mistyped language gets flagged before it reaches production tags.

Mental Model

The set is a guest list at a private club: the format of your name does not matter if it is not actually on the list.

Project 341 Validate the ISO 3166-1 Region Part

The Idea Behind It

When a region is present in an hreflang value it must be an ISO 3166-1 alpha-2 code: exactly two letters, all uppercase. This project splits each code once, then applies three conditions together so it can reject three-letter codes, lowercase regions, and the numeric UN regions that hreflang does not support, while quietly noting codes that carry no region at all.

What You'll Build & Learn

Validate the region subtag as a two-letter uppercase alpha-2 code, rejecting wrong lengths, casing, and numeric forms.

- ISO 3166-1 alpha-2 requires two uppercase letters
- Combining length, alpha, and upper checks
- Handling codes with no region gracefully

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 340: validate the iso 639-1 language part. Project 342 changes the surface problem while preserving the underlying method.

progress check

Reading target: validate the iso 3166-1 region part is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `codes = ['en-US', 'en-us', 'fr-CA', 'de-DEU', 'es-419', 'pt-BR']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

- Evidence: `print(f'{c}: region={region} {"valid" if ok else "INVALID"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`codes = ['en-US', 'en-us', 'fr-CA', 'de-DEU', 'es-419', 'pt-BR']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if '-' not in c:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{c}: region={region} {"valid" if ok else "INVALID"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if '-' not in c:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate the ISO 3166-1 Region Part: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: codes, en, US, us, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: codes = ['en-US', 'en-us', 'fr-CA', 'de-DEU', 'es-419', 'pt-BR'] | Core: if '-' not in c: | Evidence: print(f'{c}: region={region} {"valid" if ok else "INVALID"}')

🛠️ Reader to Writer Bridge

1. Read: explain validate the iso 3166-1 region part in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 340.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 342 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c}: region={region} {"valid" if ok else "INVALID"}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `codes = ['en-US', 'en-us', 'fr-CA', 'de-DEU', 'es-419', 'pt-BR']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate the iso 3166-1 region part and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

After splitting off the region, three string predicates (length 2, alphabetic, uppercase) together enforce the alpha-2 uppercase convention.

```
codes = ['en-US', 'en-us', 'fr-CA', 'de-DEU', 'es-419', 'pt-BR']
for c in codes:
    if '-' not in c:
        print(f'{c}: no region'); continue
    region = c.split('-', 1)[1]
    ok = len(region) == 2 and region.isalpha() and region.isupper()
    print(f'{c}: region={region} {"valid" if ok else "INVALID"}')
```

```
output
en-US: region=US valid
en-us: region=us INVALID
fr-CA: region=CA valid
de-DEU: region=DEU INVALID
es-419: region=419 INVALID
pt-BR: region=BR valid
```

How it works:

1. The loop first checks for a hyphen; codes without one print 'no region' and continue to the next item.
2. split('-', 1)[1] isolates everything after the first hyphen as the region string.
3. Three conditions joined with and confirm len is two, isalpha is true, and isupper is true, then the result prints as 'valid' or 'INVALID'.

Try it:

1. Check each region against a real set of ISO 3166-1 alpha-2 codes, not just the shape.
2. Allow the UN M49 region '419' as a valid Latin-America special case.
3. Report both the language and region validity for every code on one line.
4. Decide whether to allow `419` (Latin America) as a special numeric region.

⚠ Common Mistakes

- Accepting `DEU` because it is alphabetic ignores that hreflang uses two-letter alpha-2, not three-letter alpha-3.
- Using `.upper() == region` instead of `.isupper()` wrongly passes an already-lowercased value.
- Forgetting `.isalpha()` lets the numeric `419` pass the length-2 check if you only count characters.

Skills gained iso-3166-1 | string methods | validation | conditionals

→ Where You'll Use This: Region casing and length errors are among the most common hreflang mistakes; this trio of checks pinpoints them in bulk when you parse a site's alternate tags.

Project 342 Detect a Malformed Hreflang Code

The Idea Behind It

Because a search crawler silently skips any hreflang value it cannot parse, an audit must surface the broken ones rather than let them vanish. This project reuses the shape regex but inverts the logic with a list comprehension, collecting every code that fails to match and printing both the offending list and its count for a quick report.

What You'll Build & Learn

Collect and count the malformed hreflang codes from a list so an audit can report them up front.

- Filtering with a negated regex match
- List comprehensions for collecting failures
- Why crawlers drop bad codes silently

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 341: validate the iso 3166-1 region part. Project 343 changes the surface problem while preserving the underlying method.

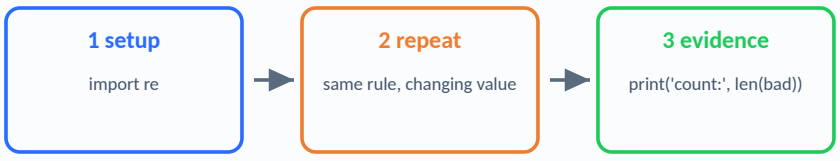
progress check

Reading target: detect a malformed hreflang code is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('count:', len(bad))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`bad = [c for c in codes if not pat.match(c)]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('count:', len(bad))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `bad = [c for c in codes if not pat.match(c)]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Malformed Hreflang Code: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: re, codes, en, US, EN. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: bad = [c for c in codes if not pat.match(c)] | Evidence: print('count:', len(bad))

🛠️ Reader to Writer Bridge

1. Read: explain detect a malformed hreflang code in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 341.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 343 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('count:', len(bad))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a malformed hreflang code and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering the list by codes that fail the canonical pattern collects every malformed value, and the count gives a quick severity gauge.

```
import re
codes = ['en-US', 'EN-us', 'e-US', 'en-USA', 'en--US', 'en']
pat = re.compile(r'^[a-z]{2}(-[A-Z]{2})?$')
bad = [c for c in codes if not pat.match(c)]
print('malformed:', bad)
print('count:', len(bad))
```

```
output
malformed: ['EN-us', 'e-US', 'en-USA', 'en--US']
count: 4
```

How it works:

1. The same '^[a-z]{2}(-[A-Z]{2})?$' pattern is compiled to define the legal shape.
2. A list comprehension keeps each c where not pat.match(c) is true, building the bad list of malformed codes.
3. Two prints report the malformed list and len(bad), giving a headline number for the report.

Try it:

1. Also return the index of each malformed code so editors can find it in the source.

2. Split the failures into 'bad casing' versus 'bad structure' buckets.
3. Write the malformed list to a CSV file for the content team.
4. Print each malformed code with the reason it failed.

⚠ Common Mistakes

- Reporting `de-de` as fine because letters match while ignoring that the region must be uppercase.
- Counting `len(codes)` instead of `len(bad)` reports total codes, not the broken ones.
- A pattern without `$` would accept `en--US` by matching only its valid prefix.

Skills gained regex | list comprehension | auditing | len

→ Where You'll Use This: Surfacing the count of malformed codes is the first metric in an hreflang health report; a non-zero number tells stakeholders tags are being dropped invisibly.

Mental Model

Malformed tags are like envelopes with no address: the postal system tosses them without a word, so you must check the outbox yourself.

Project 343 Build an Hreflang Cluster from a Page

The Idea Behind It

An hreflang cluster is the full set of language alternates for one piece of content. This project models a page's alternates as a dict mapping locale to URL, then calls sorted on its items to produce a deterministic, alphabetized view. Stable ordering matters because an audit that prints clusters the same way every time is far easier to diff and reason about.

What You'll Build & Learn

Represent a page's language alternates as a dict and print them in a stable, sorted order with a count.

- A cluster is the alternates for one URL
- Sorting dict items for deterministic output
- Counting locales in a cluster

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 342: detect a malformed hreflang code. Project 344 changes the surface problem while preserving the underlying method.

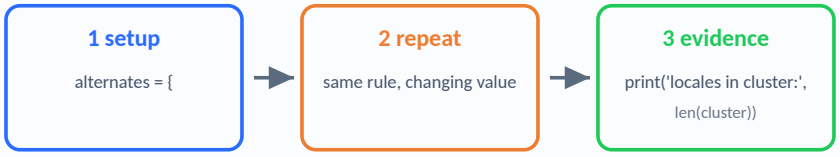
progress check

Reading target: build an hreflang cluster from a page is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `alternates = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('locales in cluster:', len(cluster))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`alternates = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`for code, url in cluster:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('locales in cluster:', len(cluster))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for code, url in cluster:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build an Hreflang Cluster from a Page: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: alternates, en, https, shop, com. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: alternates = { | Core: for code, url in cluster: | Evidence: print('locales in cluster:', len(cluster))

🛠️ Reader to Writer Bridge

1. Read: explain build an hreflang cluster from a page in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 342.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 344 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('locales in cluster:', len(cluster))`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `alternates = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build an hreflang cluster from a page and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A dict of `code -> url` is the natural cluster model; sorting its items makes the printed cluster deterministic and counting the keys gives the locale count.

```
alternates = {
    'en': 'https://shop.com/en/',
    'fr': 'https://shop.com/fr/',
    'de': 'https://shop.com/de/',
}
cluster = sorted(alternates.items())
for code, url in cluster:
    print(f'{code} -> {url}')
print('locales in cluster:', len(cluster))
```

```
output
de -> https://shop.com/de/
en -> https://shop.com/en/
fr -> https://shop.com/fr/
locales in cluster: 3
```

How it works:

1. A dict named alternates maps three language codes to their absolute URLs on shop.com.
2. sorted(alternates.items()) returns a list of code-URL tuples ordered alphabetically by code.
3. The loop prints each 'code -> url' pair, then a final line reports len(cluster) as the number of locales.

Try it:

1. Add an 'x-default' entry and decide whether to count it as a locale.
2. Sort by URL instead of by code and observe the difference.
3. Wrap the logic in a function that takes any alternates dict and returns the sorted list.
4. Add an `x-default` entry and watch the cluster size grow.

⚠ Common Mistakes

- Printing the dict directly relies on insertion order and varies the output.
- Counting characters in the dict instead of `len(cluster)` mismeasures the locale count.

- Forgetting that the page itself must appear in its own cluster (the self-reference).

Skills gained dicts sorting clusters iteration

→ Where You'll Use This: Modeling clusters as dicts is the foundation for every reciprocity and completeness check; sorted output makes audit logs stable across runs and easy to compare.

Mental Model

The cluster is a family photo: everyone who belongs together appears in one frame, and sorting just lines them up in a predictable order.

Project 344 Check Return-Tag Reciprocity

The Idea Behind It

Hreflang must be bidirectional: if page A names page B as an alternate, B has to name A in return. This project stores each page's alternates as nested dicts, then walks every alternate target and checks whether the original page appears among that target's own alternate URLs, reporting a single OK or BROKEN verdict for the whole set.

What You'll Build & Learn

Audit a small page-to-alternates map for full reciprocity and report a single pass or fail verdict.

- Hreflang reciprocity is mandatory
- Nested dicts model a cluster of pages
- A flag variable tracks overall validity

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 343: build an hreflang cluster from a page. Project 345 changes the surface problem while preserving the underlying method. *progress check*

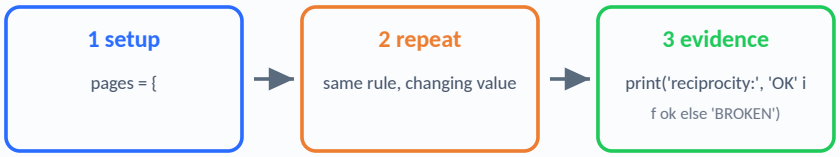
Reading target: check return-tag reciprocity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('reciprocity:', 'OK' if ok else 'BROKEN')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if page not in back.values():`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('reciprocity:', 'OK' if ok else 'BROKEN')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if page not in back.values():`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Check Return-Tag Reciprocity: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: pages, https, com, en, fr. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: if page not in back.values(): | Evidence: print('reciprocity:', 'OK' if ok else 'BROKEN')

🛠️ Reader to Writer Bridge

1. Read: explain check return-tag reciprocity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 343.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 345 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('reciprocity:', 'OK' if ok else 'BROKEN')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of check return-tag reciprocity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For each alternate target, looking up that target's own alternates and confirming the source URL appears among them proves the return tag exists.

```
pages = {
    'https://s.com/en/': {'en': 'https://s.com/en/', 'fr':
'https://s.com/fr/'},
    'https://s.com/fr/': {'en': 'https://s.com/en/', 'fr':
'https://s.com/fr/'},
}
ok = True
for page, alts in pages.items():
    for code, target in alts.items():
        back = pages.get(target, {})
        if page not in back.values():
            ok = False
            print(f'{page} -> {target} has NO return tag')
print('reciprocity:', 'OK' if ok else 'BROKEN')
```

```
output
reciprocity: OK
```

How it works:

1. The pages dict maps each page URL to its own alternates dict of code to target URL.
2. A boolean ok starts True, and nested loops visit every alternate target for every page.
3. pages.get(target, {}) fetches the target's alternates, and if the original page is not in back.values() the flag flips to False and the broken edge prints.

Try it:

1. Count the number of reciprocal edges versus total edges and print a ratio.
2. Return the list of broken pairs instead of just OK or BROKEN.
3. Add a third page and break one return tag to watch the verdict change.
4. Drop the `fr` page's `en` entry and watch reciprocity break.

⚠ Common Mistakes

- Comparing against the target's alternate keys (codes) instead of its values (URLs) never finds the return tag.
- Assuming reciprocity holds because A lists B, without checking B's tags at all.
- Using `target in back` checks dict keys, not the URL values, and gives a false negative.

Skills gained nested dicts reciprocity auditing boolean flags

→ Where You'll Use This: Reciprocity failures are the single most common reason Google ignores hreflang; this audit is the core check you run across an entire crawled cluster before launch.

Mental Model

Reciprocity is a handshake: if you extend your hand, the other person must extend theirs back, or the greeting does not count.

Project 345 Detect a Missing Reciprocal Tag

The Idea Behind It

A pass-or-fail verdict is not actionable; when reciprocity breaks, the team needs the exact missing edge. This project walks every alternate target and, for any target whose alternates do not link back to the source page, records the unreciprocated source-target pair so it can print each broken link and a total count of repairs needed.

What You'll Build & Learn

List every specific unreciprocated hreflang link so the team knows exactly which page must add a return tag.

- Reporting edges, not just a verdict
- Collecting tuples of failures
- Turning audits into fix lists

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 344: check return-tag reciprocity. Project 346 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a missing reciprocal tag is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

- Evidence: `print('broken links:', len(missing))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if page not in pages.get(target, {}).values():`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('broken links:', len(missing))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if page not in pages.get(target, {}).values():`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Detect a Missing Reciprocal Tag: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: pages, https, com, en, es. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = { | Core: if page not in pages.get(target, {}).values(): | Evidence: print('broken links:', len(missing))

🛠️ Reader to Writer Bridge

1. Read: explain detect a missing reciprocal tag in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 344.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 346 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('broken links:', len(missing))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a missing reciprocal tag and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Collecting `(source, target)` pairs whose target does not list the source pinpoints every one-way hreflang link that Google would discard.

```
pages = {
    'https://s.com/en/': {'en': 'https://s.com/en/', 'es':
'https://s.com/es/'},
    'https://s.com/es/': {'es': 'https://s.com/es/'},
}
missing = []
for page, alts in pages.items():
    for target in alts.values():
        if page not in pages.get(target, {}).values():
            missing.append((page, target))
for src, tgt in missing:
    print(f'missing return: {tgt} should link back to {src}')
print('broken links:', len(missing))
```

```
output
missing return: https://s.com/es/ should link back to https://s.com/en/
broken links: 1
```

How it works:

1. Two pages are defined where the English page links to Spanish, but the Spanish page only lists itself.
2. For each alternate target, the code checks page not in pages.get(target, {}).values() and appends a (page, target) tuple to missing when the back link is absent.
3. A loop prints 'missing return: target should link back to src' for each gap, then reports len(missing).

Try it:

1. Group missing returns by the page that needs fixing.
2. Include the language code of each missing link in the message.
3. Add a second broken pair and confirm both appear in the output.
4. Add the missing `en` entry to the `es` page and confirm zero broken links.

⚠ Common Mistakes

- Recording the target as the page to fix instead of the source that needs the return tag.
- Checking the target's alternate codes rather than its URLs misses the actual link.
- Skipping a page's self-reference can make a present return tag look absent.

Skills gained tuples reciprocity diagnostics iteration

→ Where You'll Use This: Turning a reciprocity failure into a precise list of which page must add which return tag is what makes an hreflang report something engineers can act on directly.

Project 346 Detect a Missing x-default Tag

The Idea Behind It

The x-default tag tells Google which page to serve visitors whose language matches none of your declared locales. This project stores each cluster's codes as a set and checks membership of the literal 'x-default' string, printing a clear has-or-missing line per cluster so you can spot any group that leaves international fallback to chance.

What You'll Build & Learn

Scan several clusters and flag any that lack the x-default fallback tag entirely.

- x-default handles unmatched languages
- Set membership for tag presence
- Why fallback should not be implicit

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 345: detect a missing reciprocal tag. Project 347 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a missing x-default tag is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

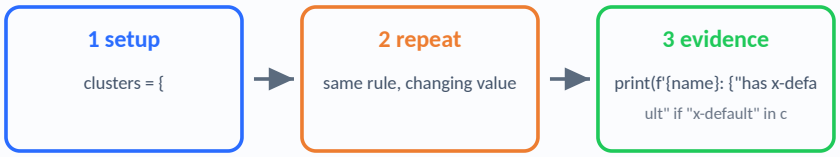
- Setup: `clusters = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING x-default"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`clusters = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING x-default"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING x-default"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING x-default"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Missing x-default Tag: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: clusters, product, en, US, GB. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: clusters = { | Core: print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING... | Evidence: print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING...

🛠️ Reader to Writer Bridge

1. Read: explain detect a missing x-default tag in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 345.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 347 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING x-default"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `clusters = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a missing x-default tag and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Testing each cluster's code set for the literal `x-default` membership flags the clusters that omit the universal fallback.

```
clusters = {
    'product-A': {'en-US', 'en-GB', 'x-default'},
    'product-B': {'en-US', 'fr-FR'},
    'product-C': {'de-DE', 'x-default'},
}
for name in sorted(clusters):
```

```
    codes = clusters[name]
    print(f'{name}: {"has x-default" if "x-default" in codes else "MISSING x-default"}')
```

```
output
product-A: has x-default
product-B: MISSING x-default
product-C: has x-default
```

How it works:

1. A dict named clusters maps product names to sets of hreflang codes, one of which omits 'x-default'.
2. Iterating sorted(clusters) keeps the report alphabetical by product.
3. An inline conditional checks 'x-default' in codes and prints 'has x-default' or 'MISSING x-default' for each.

Try it:

1. Also verify that the x-default points to a real URL, not just that the tag exists.
2. Report the percentage of clusters that are missing x-default.
3. Allow a cluster to be exempt if it serves only one global language.
4. Add `x-default` to product-B and watch it pass.

⚠ Common Mistakes

- Treating any region-less code like `en` as the default instead of the explicit `x-default`.
- Iterating the dict in insertion order makes the report nondeterministic; sort the names.
- Assuming one site-wide x-default suffices when each cluster needs its own.

Skills gained sets | x-default | membership | auditing

→ Where You'll Use This: Missing x-default is a frequent gap on multinational sites; this check ensures every cluster declares a sensible landing page for users in unsupported regions.

Mental Model

x-default is the lobby of an international building: visitors who do not know which floor they need start there instead of getting lost.

Project 347 Detect Duplicate Hreflang Entries

The Idea Behind It

Listing the same hreflang code twice on a single page is a common copy-paste mistake that confuses crawlers. This project feeds a list of code-URL tuples into a Counter keyed only on the code, then keeps any code whose tally exceeds one, surfacing duplicate declarations regardless of whether the URLs they point to agree.

What You'll Build & Learn

Use a frequency count to detect any hreflang code that is declared more than once on a page.

- Counter tallies occurrences quickly
- Duplicate codes confuse crawlers
- Generator expressions feed Counter

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 346: detect a missing x-default tag. Project 348 changes the surface problem while preserving the underlying method.

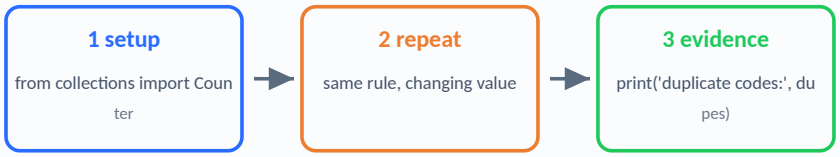
progress check

Reading target: detect duplicate hreflang entries is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('duplicate codes:', dupes)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`dupes = sorted(c for c, n in counts.items() if n > 1)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('duplicate codes:', dupes)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dupes = sorted(c for c, n in counts.items() if n > 1)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Duplicate Hreflang Entries: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: collections, Counter, entries, en, US. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: dupes = sorted(c for c, n in counts.items() if n > 1) | Evidence: print('duplicate codes:', dupes)

🛠️ Reader to Writer Bridge

1. Read: explain detect duplicate hreflang entries in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 346.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 348 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('duplicate codes:', dupes)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect duplicate hreflang entries and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `Counter` over the codes tallies occurrences, and filtering for counts above one collects the duplicated codes deterministically once sorted.

```
from collections import Counter
entries = [
    ('en-US', 'https://s.com/en/'),
    ('fr-FR', 'https://s.com/fr/'),
    ('en-US', 'https://s.com/en2/'),
    ('de-DE', 'https://s.com/de/'),
]
counts = Counter(code for code, _ in entries)
dupes = sorted(c for c, n in counts.items() if n > 1)
print('duplicate codes:', dupes)
```

```
output
duplicate codes: ['en-US']
```

How it works:

1. A list named entries holds (code, url) tuples, with 'en-US' appearing twice pointing at different URLs.
2. Counter(code for code, _ in entries) counts how often each code appears, ignoring the URL via the throwaway underscore.
3. A comprehension keeps codes where n > 1, sorts them, and prints the duplicate list.

Try it:

1. Report how many times each duplicate code appears, not just that it repeats.
2. List the conflicting URLs alongside each duplicate code.
3. Ignore duplicates that point at the identical URL as harmless.
4. Add a second `de-DE` entry and watch it join the duplicate list.

⚠ Common Mistakes

- Counting `(code, url)` pairs treats two different URLs under one code as distinct and hides the duplicate code.
- Using a set to dedupe loses the information that a code repeated at all.
- Forgetting to sort the duplicate list makes the output order unstable.

Skills gained Counter duplicates generators collections

→ Where You'll Use This: Duplicate-code detection is a standard line item in tag QA; a content management system that injects alternates twice produces exactly this defect, and the count catches it.

⚡ Mental Model

A duplicate code is calling two people by the same name at a meeting: the room does not know which one you mean.

Project 348 Detect Conflicting Hreflang Targets

💡 The Idea Behind It

A harmless duplicate points to one URL twice, but a conflict is worse: one hreflang code mapped to two different URLs. This project builds a defaultdict of sets so each code accumulates its distinct targets, then any code whose set holds more than one URL is reported as a CONFLICT, the situation where Google cannot pick a winner and may drop the tag.

What You'll Build & Learn

Group alternate entries by code and flag any code that resolves to more than one distinct URL.

- Conflicts differ from plain duplicates
- defaultdict(set) collects unique targets
- Set size reveals ambiguity

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 347: detect duplicate hreflang entries. Project 349 changes the surface problem while preserving the underlying method.

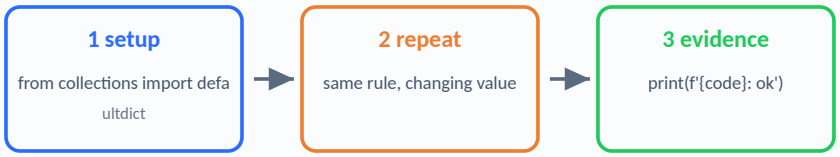
progress check

Reading target: detect conflicting hreflang targets is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{code}: ok')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if len(urls) > 1:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{code}: ok')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if len(urls) > 1:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Conflicting Hreflang Targets: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: collections, defaultdict, entries, en, US. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: if len(urls) > 1: | Evidence: print(f'{code}: ok')

🛠️ Reader to Writer Bridge

1. Read: explain detect conflicting hreflang targets in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 347.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 349 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code}: ok')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect conflicting hreflang targets and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Grouping URLs into a set per code reveals a conflict whenever a code's URL set has more than one member, while a single-member set is a clean mapping.

```
from collections import defaultdict
entries = [
    ('en-US', 'https://s.com/en/'),
    ('en-US', 'https://s.com/us/'),
    ('fr-FR', 'https://s.com/fr/'),
]
by_code = defaultdict(set)
for code, url in entries:
    by_code[code].add(url)
for code in sorted(by_code):
    urls = by_code[code]
    if len(urls) > 1:
        print(f'{code}: CONFLICT -> {sorted(urls)}')
    else:
        print(f'{code}: ok')
```

```
output
en-US: CONFLICT -> ['https://s.com/en/', 'https://s.com/us/']
fr-FR: ok
```

How it works:

1. A defaultdict(set) named by_code lets each new code start with an empty set automatically.
2. The loop adds each url to by_code[code], so duplicate URLs collapse but distinct ones accumulate.
3. For each code, len(urls) > 1 marks a CONFLICT and prints the sorted list of clashing URLs, else it prints 'ok'.

Try it:

1. Count the number of conflicting codes and exit non-zero if any exist.
2. Suggest the most frequently seen URL as the likely correct target.
3. Distinguish conflicts caused by trailing-slash differences only.
4. Point both `en-US` entries at the same URL and watch the conflict vanish.

⚠ Common Mistakes

- Using a list instead of a set reports a benign exact-duplicate URL as a conflict.
- Reporting the first URL and ignoring the rest hides that a conflict exists at all.
- Iterating `by_code` without sorting makes the conflict report nondeterministic.

Skills gained `defaultdict` `sets` `conflicts` `grouping`

→ **Where You'll Use This:** Conflicting targets often arise when a redirect or canonical change leaves stale alternates behind; spotting them prevents Google from discarding the entire hreflang block.

⚡ Mental Model

A conflict is two signposts at a fork both labeled 'this way': the traveller stops because neither can be trusted.

Project 349 Verify Self-Canonical Within a Locale

💡 The Idea Behind It

Each localized page should canonicalize to itself, not to another language version, so that every locale stays eligible for indexing. This project maps each page URL to its declared canonical and compares the two: when they match it prints 'self', and when a page points its canonical at a different locale it flags that the canonical points away.

What You'll Build & Learn

Check that each localized page declares itself as its own canonical rather than pointing at another locale.

- Localized pages need self-canonicals
- Cross-locale canonicals deindex versions
- Comparing a page to its canonical

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 348: detect conflicting hreflang targets. Project 350 changes the surface problem while preserving the underlying method.

progress check

Reading target: verify self-canonical within a locale is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

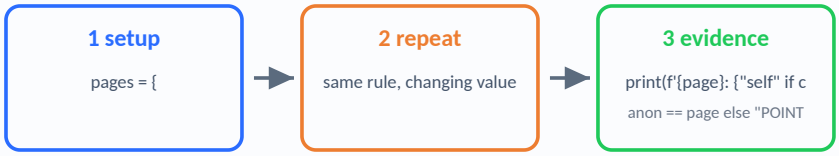
🔎 Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Verify Self-Canonical Within a Locale: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: pages, https, com, en, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon... | Evidence: print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon...

🛠️ Reader to Writer Bridge

1. Read: explain verify self-canonical within a locale in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 348.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 350 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of verify self-canonical within a locale and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Comparing each page's canonical to its own URL flags any page that hands its indexing signal to a different locale.

```
pages = {
    'https://s.com/en/': 'https://s.com/en/',
    'https://s.com/fr/': 'https://s.com/fr/',
    'https://s.com/de/': 'https://s.com/en/',
}
for page in sorted(pages):
    canon = pages[page]
    print(f'{page}: {"self" if canon == page else "POINTS AWAY -> " + canon}')
```

```
output
https://s.com/de/: POINTS AWAY -> https://s.com/en/
https://s.com/en/: self
https://s.com/fr/: self
```

How it works:

1. A dict named pages maps each page URL to its declared canonical URL, with the German page wrongly pointing at English.
2. Iterating sorted(pages) gives a stable order over the page URLs.
3. The check canon == page prints 'self' when they agree, otherwise it prints 'POINTS AWAY ->' followed by the foreign canonical.

Try it:

1. Treat a canonical that differs only by trailing slash as self-referential.
2. Report the count of pages canonicalizing away from themselves.
3. Fetch the live canonical tag instead of reading from a static dict.
4. Fix the `de` page's canonical to point at itself and confirm all 'self'.

⚠ Common Mistakes

- Treating a canonical that points to the default locale as acceptable de-indexes the other languages.
- Comparing only the path, ignoring host, can mask a cross-domain canonical.
- Iterating the dict unsorted makes the per-page report order unstable.

Skills gained canonicals dicts comparison indexing

→ Where You'll Use This: A non-self canonical on a localized page is a silent indexing killer; this check is run alongside hreflang audits because the two signals must agree for international pages to rank.

Mental Model

A self-canonical is signing your own name on your own work; pointing it elsewhere is crediting someone else for what you made.

Project 350 Flag a Cross-Locale Canonical Error

The Idea Behind It

Beyond pointing to itself, a canonical must stay within its own locale. This project keeps a locale map and a canonical map, then compares the locale of each page against the locale of its canonical target. When the French page canonicalizes to English the locales differ, so it prints WRONG, capturing the precise error that tells Google to drop a language from the index.

What You'll Build & Learn

Detect a canonical that crosses locale boundaries by comparing the locale of the page and its canonical target.

- Canonicals must stay within a locale
- Mapping URLs to their locale
- Comparing two lookups to find errors

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 349: verify self-canonical within a locale. Project 351 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag a cross-locale canonical error is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `locale_of = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

- Evidence: `print(f'{page}: same-locale canonical OK')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`locale_of = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if locale_of[page] != locale_of.get(canon):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{page}: same-locale canonical OK')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if locale_of[page] != locale_of.get(canon):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag a Cross-Locale Canonical Error: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: locale_of, https, com, en, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: locale_of = { | Core: if locale_of[page] != locale_of.get(canon): | Evidence: print(f'{page}: same-locale canonical OK')

🛠️ Reader to Writer Bridge

1. Read: explain flag a cross-locale canonical error in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 349.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 351 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{page}: same-locale canonical OK')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `locale_of = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag a cross-locale canonical error and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Looking up the locale of both the page and its canonical target and comparing them detects when a canonical crosses a language boundary.

```
locale_of = {
    'https://s.com/en/': 'en',
    'https://s.com/fr/': 'fr',
}
canonicals = {
    'https://s.com/en/': 'https://s.com/en/',
    'https://s.com/fr/': 'https://s.com/en/',
}
for page in sorted(canonicals):
    canon = canonicals[page]
    if locale_of[page] != locale_of.get(canon):
        print(f'{page} ({locale_of[page]}) canonicalises to
{locale_of.get(canon)} — WRONG')
    else:
        print(f'{page}: same-locale canonical OK')
```

```
output
https://s.com/en/: same-locale canonical OK
https://s.com/fr/ (fr) canonicalises to en — WRONG
```

How it works:

1. locale_of maps each page URL to its language, and canonicals maps each page to its declared canonical.
2. For each page, the code compares locale_of[page] against locale_of.get(canon), the locale of the canonical target.
3. When the two locales differ it prints a WRONG message naming both locales, otherwise it confirms a same-locale canonical.

Try it:

1. Handle a canonical target missing from locale_of without raising a KeyError.
2. Print the page's URL plus the offending canonical URL for both sides.
3. Aggregate which locale is most often the wrong canonical destination.
4. Repoint the `fr` page's canonical to itself and watch the error clear.

⚠ Common Mistakes

- Comparing the raw URLs instead of their locales misses that a same-locale alias is fine.
- Using `locale_of[canon]` (not `.get`) raises a KeyError when the canonical is an unknown URL.
- Assuming any self-pointing canonical is correct without checking the locale mapping behind it.

Skills gained canonicals | locales | dict lookups | diagnostics

→ Where You'll Use This: Cross-locale canonicalization quietly removes whole languages from search; pairing a locale map with a canonical map is how an audit proves each version canonicalizes inside its own language.

⚡ Mental Model

Each locale is a separate filing drawer; a cross-locale canonical files the French folder inside the English drawer, so French disappears.

Project 351 Count Distinct Locales in a Cluster

💡 The Idea Behind It

An audit summary needs two numbers: how many locale variants you serve and how many distinct languages sit behind them. This project filters out the 'x-default' entry because it is a fallback rather than a locale, counts the remaining variants, and derives the unique set of language subtags by splitting each code on its hyphen.

What You'll Build & Learn

Summarize a cluster by counting locale variants and the distinct languages behind them, excluding x-default.

- Locale variants versus distinct languages
- x-default is a fallback, not a locale
- Set comprehension for unique languages

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 350: flag a cross-locale canonical error. Project 352 changes the surface problem while preserving the underlying method.

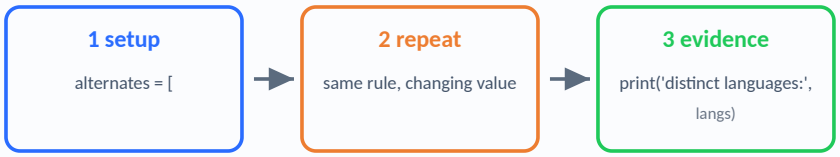
progress check

Reading target: count distinct locales in a cluster is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `alternates = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('distinct languages:', langs)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`alternates = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`locales = [c for c in alternates if c != 'x-default']`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('distinct languages:', langs)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `locales = [c for c in alternates if c != 'x-default']`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Distinct Locales in a Cluster: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: alternates, en, US, GB, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: alternates = [| Core: locales = [c for c in alternates if c != 'x-default'] | Evidence: print('distinct languages:', langs)

🛠️ Reader to Writer Bridge

1. Read: explain count distinct locales in a cluster in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 350.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 352 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('distinct languages:', langs)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `alternates = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count distinct locales in a cluster and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering out `x-default` leaves true locale variants, and reducing each to its language subtag via a set counts the distinct languages served.

```
alternates = [
    'en-US', 'en-GB', 'fr-FR', 'fr-CA', 'de-DE', 'x-default',
]
locales = [c for c in alternates if c != 'x-default']
langs = sorted({c.split('-')[0] for c in locales})
print('locale variants:', len(locales))
print('distinct languages:', langs)
```

```
output
locale variants: 5
distinct languages: ['de', 'en', 'fr']
```

How it works:

1. A list named alternates holds five region variants plus the 'x-default' token.
2. A comprehension builds locales by keeping every code that is not 'x-default'.
3. A set comprehension over c.split('-')[0] collects the unique language parts, and the two prints report the variant count and the sorted languages.

Try it:

1. Also report how many regions each language serves.
2. Raise the count if 'x-default' is missing from the list.
3. Sort the language list by number of regions instead of alphabetically.
4. Add `es-ES` and watch both counts rise.

⚠ Common Mistakes

- Counting `x-default` as a locale inflates the variant total by one.
- Counting locale variants as if they were distinct languages double-counts `en-US` and `en-GB`.
- Forgetting to sort the language set makes the printed list nondeterministic.

Skills gained set comprehension | counting | string split | x-default

→ Where You'll Use This: Reporting both variant count and language count gives stakeholders an honest picture of international coverage, distinguishing many regions of one language from genuine multilingual reach.

⚡ Mental Model

It is the difference between counting cars and counting car brands: five variants might all be the same language wearing different plates.

Project 352 Check Hreflang Matrix Completeness

💡 The Idea Behind It

In a complete hreflang matrix every locale links to all locales in the set, including itself. This project holds an expected set of languages and a matrix mapping each locale to the set it actually links to, then uses set subtraction to find what each locale is missing, reporting the German page's omission of French.

What You'll Build & Learn

Verify that every locale in a matrix links to the full expected set, reporting each locale's missing alternates.

- A complete matrix links every locale to all
- Set difference finds gaps
- One gap weakens the whole cluster

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 351: count distinct locales in a cluster. Project 353 changes the surface problem while preserving the underlying method.

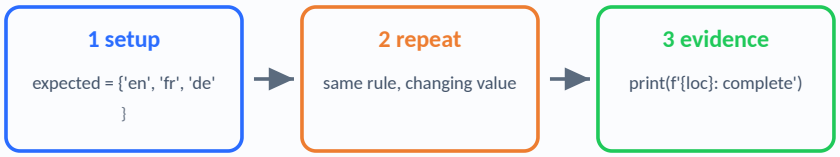
progress check

Reading target: check hreflang matrix completeness is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `expected = {'en', 'fr', 'de'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{loc}: complete')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`expected = {'en', 'fr', 'de'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

Move	Code to inspect	Why it matters
2. Apply the rule	`if missing:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: complete')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if missing:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Hreflang Matrix Completeness: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: expected, en, fr, de, matrix. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: expected = {'en', 'fr', 'de'} | Core: if missing: | Evidence: print(f'{loc}: complete')

Reader to Writer Bridge

1. Read: explain check hreflang matrix completeness in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 351.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 353 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: complete')`. Then compare punctuation, spacing, and order.
2. Explain: point to `expected = {'en', 'fr', 'de'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check hreflang matrix completeness and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Subtracting each locale's declared set from the expected full set reveals exactly which alternates that page is missing.

```
expected = {'en', 'fr', 'de'}
matrix = {
    'en': {'en', 'fr', 'de'},
    'fr': {'en', 'fr', 'de'},
    'de': {'en', 'de'},
}
for loc in sorted(matrix):
    missing = expected - matrix[loc]
    if missing:
        print(f'{loc}: missing {sorted(missing)}')
    else:
        print(f'{loc}: complete')
```

```
output
de: missing ['fr']
en: complete
fr: complete
```

How it works:

1. An expected set lists the three languages that should appear, and matrix maps each locale to the set it actually links to.
2. For each locale, expected - matrix[loc] computes the set of languages it is missing.
3. If that difference is non-empty the code prints the sorted missing list, otherwise it prints 'complete'.

Try it:

1. Also detect extra locales a page links to that are not in expected.
2. Print a full grid showing every locale-to-locale link status.
3. Compute a completeness percentage across the whole matrix.
4. Add the missing `fr` to the `de` row and watch it report complete.

⚠ Common Mistakes

- Checking only that each page links to the others while forgetting the required self-reference.
- Comparing list lengths instead of set differences hides which specific locale is absent.
- Iterating the matrix unsorted makes the per-locale report order vary.

Skills gained set difference | matrix | completeness | auditing

→ Where You'll Use This: Matrix completeness checks catch the case where one page in a cluster forgot an alternate; that single gap breaks reciprocity for the missing pair and undermines the entire group.

Mental Model

The matrix is a round-robin tournament bracket: if one player skips a match, the whole schedule is incomplete.

Project 353 Detect an Orphan Alternate URL

The Idea Behind It

An hreflang tag can point at a URL that does not actually exist on the site, an orphan alternate that wastes crawl budget and breaks reciprocity. This project keeps a set of URLs that really exist, then filters the linked alternate URLs to find any that are absent from that set, surfacing the Italian page that was declared but never built.

What You'll Build & Learn

Find alternate URLs declared in hreflang that do not exist among the site's real pages.

- Orphan alternates point at nonexistent URLs
- Set membership detects missing pages
- Orphans break reciprocity and waste crawl

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 352: check hreflang matrix completeness. Project 354 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect an orphan alternate url is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `existing = {'https://s.com/en/', 'https://s.com/fr/'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('orphan alternates:', orphans)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`existing = {'https://s.com/en/', 'https://s.com/fr/'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`orphans = sorted(u for u in linked.values() if u not in existing)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('orphan alternates:', orphans)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `orphans = sorted(u for u in linked.values() if u not in existing)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect an Orphan Alternate URL: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: existing, https, com, en, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: existing = {'https://s.com/en/', 'https://s.com/fr/'} | Core: orphans = sorted(u for u in linked.values() if u not in existing) | Evidence: print('orphan alternates:', orphans)

🛠️ Reader to Writer Bridge

1. Read: explain detect an orphan alternate url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 352.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 354 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('orphan alternates:', orphans)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `existing = {'https://s.com/en/', 'https://s.com/fr/'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect an orphan alternate url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Comparing each linked alternate URL against the set of pages that truly exist isolates the alternates whose targets are absent.

```
existing = {'https://s.com/en/', 'https://s.com/fr/'}
linked = {
    'en': 'https://s.com/en/',
    'fr': 'https://s.com/fr/',
    'it': 'https://s.com/it/',
}
orphans = sorted(u for u in linked.values() if u not in existing)
print('orphan alternates:', orphans)
```

```
output
orphan alternates: ['https://s.com/it/']
```

How it works:

1. A set named existing lists the URLs that truly exist on the site.
2. A dict named linked maps locales to declared alternate URLs, including an Italian URL that is not in existing.
3. A comprehension keeps every linked URL not in existing, sorts the result, and prints the orphan alternates list.

Try it:

1. Report which locale code declared each orphan URL.
2. Cross-check orphans against a live status-code crawl for 404s.
3. Add several orphans and confirm all appear sorted.
4. Add `https://s.com/it/` to the existing set and confirm zero orphans.

⚠ Common Mistakes

- Comparing alternate codes against existing URLs instead of comparing the URLs themselves.
- Treating a relative alternate as missing when it merely needs resolving against the host first.
- Forgetting to sort the orphan list makes the output order unstable.

Skills gained sets orphans membership crawl budget

→ Where You'll Use This: Orphan alternates appear when a locale is deprecated but its hreflang tags linger; finding them stops crawlers chasing dead URLs and prevents reciprocity from silently failing.

Mental Model

An orphan alternate is a phone number on a business card for a line that was disconnected: dial it and you reach nothing.

Project 354 Catch a Lang and Region Case Swap

The Idea Behind It

A classic hreflang typo inverts the casing, writing an uppercase language and a lowercase region as in 'EN-us'. This project partitions each code on its first hyphen and checks the case of the language and region subtags independently, collecting a precise list of issues per code so the editor sees exactly which half is wrong.

What You'll Build & Learn

Detect case mistakes by checking the language and region subtags separately and reporting each specific issue.

- Language should be lowercase, region uppercase
- str.partition splits on first hyphen
- Reporting each issue separately

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 353: detect an orphan alternate url. Project 355 changes the surface problem while preserving the underlying method.

progress check

Reading target: catch a lang and region case swap is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `codes = ['en-US', 'EN-us', 'fr-FR', 'DE-de', 'es-ES']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

- Evidence: `print(f'{c}: {"; ".join(issues) if issues else "ok"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`codes = ['en-US', 'EN-us', 'fr-FR', 'DE-de', 'es-ES']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if not lang.islower():`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{c}: {"; ".join(issues) if issues else "ok"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not lang.islower():`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Catch a Lang and Region Case Swap: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: codes, en, US, EN, us. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: codes = ['en-US', 'EN-us', 'fr-FR', 'DE-de', 'es-ES'] | Core: if not lang.islower(): | Evidence: print(f'{c}: {"; ".join(issues) if issues else "ok"}')

🛠️ Reader to Writer Bridge

1. Read: explain catch a lang and region case swap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 353.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 355 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c}: {"; ".join(issues) if issues else "ok"}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `codes = ['en-US', 'EN-us', 'fr-FR', 'DE-de', 'es-ES']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of catch a lang and region case swap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Partitioning on `-` yields the language and region, and `islower()`/`isupper()` on each subtag reports precisely which half has the wrong case.

```
codes = ['en-US', 'EN-us', 'fr-FR', 'DE-de', 'es-ES']
for c in codes:
    lang, _, region = c.partition('-')
    issues = []
    if not lang.islower():
        issues.append('lang should be lowercase')
    if region and not region.isupper():
        issues.append('region should be uppercase')
    print(f'{c}: {"; ".join(issues) if issues else "ok"}')
```

```
output
en-US: ok
EN-us: lang should be lowercase; region should be uppercase
fr-FR: ok
DE-de: lang should be lowercase; region should be uppercase
es-ES: ok
```

How it works:

1. c.partition('-') splits each code into lang, the separator, and region in one call.
2. If lang.islower() is false an issue is appended, and if a region exists but is not isupper another issue is appended.
3. The issues list is joined with semicolons and printed, or 'ok' is shown when the list is empty.

Try it:

1. Auto-correct the casing and print the fixed code beside the issue.
2. Treat a code with no region but uppercase language as still wrong.
3. Count how many codes have at least one casing issue.
4. Add `en-Us` (only the region's second letter lowercased) and see it flagged.

⚠ Common Mistakes

- Lowercasing the whole code before checking erases the very casing error you are hunting.
- Using `partition('-')` is safe for a region-less code, but indexing `[1]` after `split` would raise.
- Reporting a single generic 'invalid' loses which subtag was miscased.

Skills gained partition | case checks | diagnostics | hreflang

→ Where You'll Use This: Case-swap typos slip past structural shape checks because the letters are right; checking each subtag's case separately is how a thorough audit catches the most embarrassing common mistake.

Mental Model

It is a typo in someone's title: 'mr SMITH' has both halves wrong, and you need to point at each separately to fix it.

Project 355 Resolve a Language Fallback Chain

The Idea Behind It

When a requested locale is not served, negotiation falls back: from a full locale like 'en-US' to the bare language 'en', then to 'x-default'. This project defines a resolve function that builds that ordered chain for any request and returns the first candidate present in the available set, modeling how a browser or server settles on a locale.

What You'll Build & Learn

Model locale negotiation by resolving a requested code down a fallback chain to whatever is actually available.

- Fallback chain locale, language, x-default
- First match wins in negotiation
- Returning none when nothing matches

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 354: catch a lang and region case swap. Project 356 changes the surface problem while preserving the underlying method.

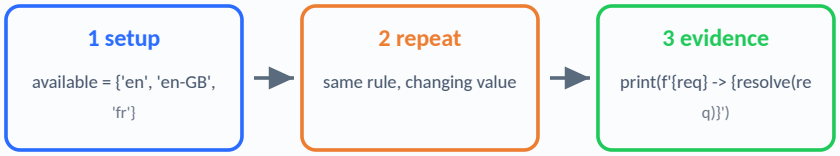
progress check

Reading target: resolve a language fallback chain is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `available = {'en', 'en-GB', 'fr'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{req} -> {resolve(req)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`available = {'en', 'en-GB', 'fr'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if cand in available:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{req} -> {resolve(req)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if cand in available:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Resolve a Language Fallback Chain: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: available, en, GB, fr, resolve. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: available = {'en', 'en-GB', 'fr'} | Core: if cand in available: | Evidence: print(f'{req} -> {resolve(req)}')

🛠️ Reader to Writer Bridge

1. Read: explain resolve a language fallback chain in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 354.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 356 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{req} -> {resolve(req)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `available = {'en', 'en-GB', 'fr'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of resolve a language fallback chain and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Building the candidate chain `[exact, language, x-default]` and returning the first one present in the available set implements graceful locale degradation.

```
available = {'en', 'en-GB', 'fr'}
def resolve(req):
    chain = [req, req.split('-')[0], 'x-default']
    for cand in chain:
        if cand in available:
            return cand
    return 'none'
```

```
for req in ['en-GB', 'en-US', 'fr-CA', 'de-DE']:
    print(f'{req} -> {resolve(req)}')
```

```
output
en-GB -> en-GB
en-US -> en
fr-CA -> fr
de-DE -> none
```

How it works:

1. The available set names the locales actually served, such as 'en', 'en-GB', and 'fr'.
2. resolve builds chain as [req, req.split('-')[0], 'x-default'] and returns the first candidate found in available.
3. Calling resolve on several requests prints how each one degrades, with 'de-DE' resolving to 'none' since no German fallback exists.

Try it:

1. Add a region-to-language map so 'fr-CA' can prefer 'fr-FR' before bare 'fr'.
2. Log the full chain that was tried for each request.
3. Return the matched URL instead of just the matched code.
4. Add `de` to the available set and watch `de-DE` resolve.

⚠ Common Mistakes

- Returning the exact request without a fallback chain leaves unmatched users with nothing.
- Putting `x-default` before the bare language wrongly skips a perfectly good language match.
- Splitting on `_` instead of `-` never extracts the bare-language fallback.

Skills gained functions fallback locales negotiation

→ Where You'll Use This: Server-side locale routing relies on exactly this degradation logic; understanding it helps you predict which page a visitor lands on and why some requests fall through to a default.

Mental Model

Locale negotiation is asking for a specific coffee, then a generic one, then 'whatever you have' until the barista can actually pour something.

Project 356 Require Absolute Alternate URLs

The Idea Behind It

Hreflang alternates must be fully-qualified absolute URLs; search engines ignore relative paths entirely. This project keeps an alternates dict mixing absolute and relative values, then tests each URL with a simple

startswith('https://') check, labeling the absolute ones valid and printing the broken relative paths so they can be corrected.

What You'll Build & Learn

Separate absolute alternate URLs from relative ones that search engines will ignore.

- Alternates must be absolute URLs
- Relative paths are silently ignored
- startswith is a cheap scheme check

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 355: resolve a language fallback chain. Project 357 changes the surface problem while preserving the underlying method.

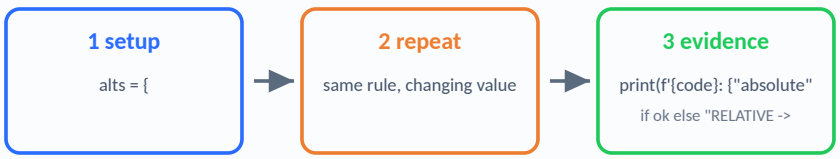
progress check

Reading target: require absolute alternate urls is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `alts = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`alts = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Require Absolute Alternate URLs: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: alts, en, https, com, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: alts = { | Core: print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}') | Evidence: print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}')

🛠️ Reader to Writer Bridge

1. Read: explain require absolute alternate urls in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 355.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 357 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `alts = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of require absolute alternate urls and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Testing each alternate for a `https://` prefix flags relative paths like `/fr/` and `es/index.html` that crawlers cannot resolve from an hreflang tag.

```
alts = {
    'en': 'https://s.com/en/',
    'fr': '/fr/',
    'de': 'https://s.com/de/',
    'es': 'es/index.html',
}
for code in sorted(alts):
    url = alts[code]
    ok = url.startswith('https://')
    print(f'{code}: {"absolute" if ok else "RELATIVE -> " + url}')
```

```
output
de: absolute
en: absolute
es: RELATIVE -> es/index.html
fr: RELATIVE -> /fr/
```

How it works:

1. A dict named alts maps locales to URLs, two of which are relative paths like '/fr/' and 'es/index.html'.

2. Iterating sorted(alts) keeps the output ordered by locale code.
3. url.startswith('https://') decides each case, printing 'absolute' or 'RELATIVE ->' with the offending value.

Try it:

1. Also accept http:// but warn that https is preferred.
2. Resolve relative paths against a base URL and print the absolute form.
3. Reject protocol-relative '//host/path' URLs as ambiguous.
4. Make every alternate absolute and confirm they all pass.

⚠ Common Mistakes

- Accepting a root-relative `/fr/` because it 'looks like a URL' though hreflang requires absolute form.
- Checking only for a leading `/` would still reject a valid absolute `https://` URL.
- Iterating the dict unsorted makes the per-code report order vary.

Skills gained startswith | absolute urls | validation | dicts

→ Where You'll Use This: Templating bugs frequently emit relative alternate URLs that search engines discard; a scheme check across every alternate is a fast guard before tags ship to production.

⚡ Mental Model

A relative URL is directions that say 'turn left at the shop' with no city named: useless to anyone who is not already standing where you are.

Project 357 Reconcile Head and Sitemap Hreflang

💡 The Idea Behind It

Hreflang can be declared in the page head or in the XML sitemap, but the two sources must agree. This project represents each source as a set of locale codes and uses set arithmetic to find which locales appear only in the head, which appear only in the sitemap, and whether the two sets are equal overall.

What You'll Build & Learn

Reconcile head-based and sitemap-based hreflang by finding the locales unique to each source.

- Hreflang lives in head or sitemap
- The two sources must match
- Set difference and equality compare them

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 356: require absolute alternate urls. Project 358 changes the surface problem while preserving the underlying method.

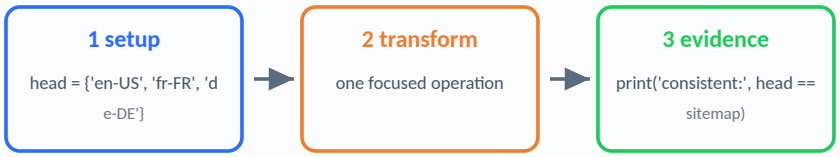
progress check

Reading target: reconcile head and sitemap hreflang is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `head = {'en-US', 'fr-FR', 'de-DE'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is deliberately small: Python stores a value, applies one clear operation, and prints evidence. That tiny chain is the reading pattern larger programs still use.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('consistent:', head == sitemap)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`head = {'en-US', 'fr-FR', 'de-DE'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`sitemap = {'en-US', 'fr-FR', 'es-ES'}`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print('consistent:', head == sitemap)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sitemap = {'en-US', 'fr-FR', 'es-ES'}`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Reconcile Head and Sitemap Hreflang: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: head, en, US, fr, FR. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: head = {'en-US', 'fr-FR', 'de-DE'} | Core: sitemap = {'en-US', 'fr-FR', 'es-ES'} | Evidence: print('consistent:', head == sitemap)

🛠️ Reader to Writer Bridge

1. Read: explain reconcile head and sitemap hreflang in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 356.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 358 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('consistent:', head == sitemap)`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `head = {'en-US', 'fr-FR', 'de-DE'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of reconcile head and sitemap hreflang and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Subtracting the sitemap set from the head set (and vice versa) reveals locale declarations that exist in just one source, and equality confirms full consistency.

```
head = {'en-US', 'fr-FR', 'de-DE'}
sitemap = {'en-US', 'fr-FR', 'es-ES'}
print('in head only:', sorted(head - sitemap))
print('in sitemap only:', sorted(sitemap - head))
print('consistent:', head == sitemap)
```

```
output
in head only: ['de-DE']
in sitemap only: ['es-ES']
consistent: False
```

How it works:

1. Two sets, head and sitemap, hold the locale codes each source declares.
2. head - sitemap finds locales only in the head and sitemap - head finds those only in the sitemap.
3. head == sitemap prints whether the two declarations are fully consistent.

Try it:

1. Report the intersection as the locales both sources agree on.
2. Extend to compare URLs per locale, not just the code set.
3. Run the comparison across many pages and total the inconsistencies.
4. Add `es-ES` to the head and watch the 'sitemap only' list shrink.

⚠ Common Mistakes

- Comparing the two as lists lets ordering differences masquerade as content differences.
- Reporting only the union hides which source is missing a locale.
- Assuming the head is authoritative without flagging sitemap-only locales.

Skills gained sets | set difference | reconciliation | sitemaps

→ **Where You'll Use This:** Large sites often manage head tags and sitemap entries with different systems; a drift between them confuses crawlers, and this set comparison is how you prove the two stay in sync.

Mental Model

The head and the sitemap are two witnesses to the same event; if their stories differ, the investigator notes exactly where they diverge.

Project 358 Require an Hreflang Self-Reference

The Idea Behind It

Every page in a cluster must list its own locale among its hreflang alternates, or the whole set is invalidated for that page. This project pairs a pages dict of alternate sets with a locale map, then checks for each page whether its own locale appears in its alternate set, confirming the required self-reference.

What You'll Build & Learn

Confirm that each page includes its own locale in its hreflang alternates, the mandatory self-reference.

- Each page must self-reference
- A missing self-ref voids the set
- Pairing alternates with a locale map

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 357: reconcile head and sitemap hreflang. Project 359 changes the surface problem while preserving the underlying method.

progress check

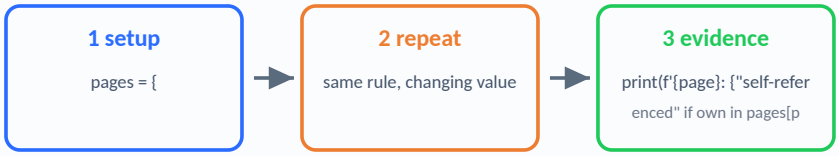
Reading target: require an hreflang self-reference is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

- Evidence: `print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING self ref"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING self ref"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING self ref"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING self ref"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Require an Hreflang Self-Reference: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: pages, https, com, en, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING... | Evidence: print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING...

🛠️ Reader to Writer Bridge

1. Read: explain require an hreflang self-reference in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 357.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 359 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING self ref"}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of require an hreflang self-reference and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Looking up each page's own locale and checking it is present in that page's declared code set enforces the self-reference rule.

```
pages = {
    'https://s.com/en/': {'en', 'fr'},
    'https://s.com/fr/': {'en', 'fr'},
}
locale = {'https://s.com/en/': 'en', 'https://s.com/fr/': 'fr'}
for page in sorted(pages):
    own = locale[page]
    print(f'{page}: {"self-referenced" if own in pages[page] else "MISSING self ref"}')
```

```
output
https://s.com/en/: self-referenced
https://s.com/fr/: self-referenced
```

How it works:

1. The pages dict maps each URL to the set of locale codes it declares as alternates.
2. A separate locale dict records which locale each page actually is.
3. For each page, own in pages[page] checks whether its own locale is listed, printing 'self-referenced' or 'MISSING self ref'.

Try it:

1. Report the count of pages missing their self-reference.
2. Verify the self-reference URL actually points at the page itself.
3. Combine this with the reciprocity check into one report.
4. Remove `en` from the `en` page's set and watch it fail.

⚠ Common Mistakes

- Assuming the self-reference is implicit and never declaring it in the tags.
- Checking for the page URL among codes instead of the page's locale subtag.
- Iterating the pages unsorted makes the per-page verdict order unstable.

Skills gained sets self-reference dicts validation

→ Where You'll Use This: Google requires the self-referential alternate; a page that lists its siblings but forgets itself has its entire hreflang block disregarded, so this check guards a non-obvious but critical rule.

⚡ Mental Model

Forgetting your self-reference is leaving yourself off the group roster you wrote: the team is told you are not on it.

Project 359 Normalize Hreflang Code Casing

The Idea Behind It

Crawlers tolerate any casing, but storing codes in one canonical form, lowercase language and uppercase region, makes deduplication and comparison reliable. This project defines a norm function that partitions each raw code, lowercases the language, and conditionally appends an uppercased region, turning messy input like 'EN-us' and 'Fr-fr' into a tidy standard shape.

What You'll Build & Learn

Normalize mixed-case hreflang codes into a canonical lowercase-language, uppercase-region form.

- Canonical form aids deduplication
- Lowercase language, uppercase region
- Conditionally rebuilding the region part

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 358: require an hreflang self-reference. Project 360 changes the surface problem while preserving the underlying method.

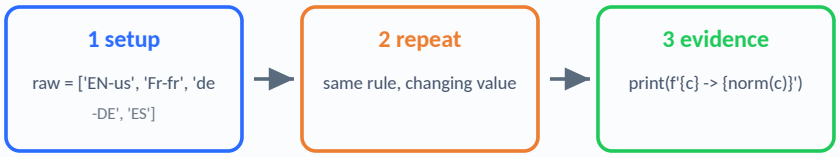
progress check

Reading target: normalize hreflang code casing is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `raw = ['EN-us', 'Fr-fr', 'de-DE', 'ES']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{c} -> {norm(c)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`raw = ['EN-us', 'Fr-fr', 'de-DE', 'ES']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`return lang.lower() + ('-' + region.upper() if region else '')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{c} -> {norm(c)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return lang.lower() + ('-' + region.upper() if region else '')`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Normalize Hreflang Code Casing: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: raw, EN, us, Fr, fr. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: raw = ['EN-us', 'Fr-fr', 'de-DE', 'ES'] | Core: return lang.lower() + ('-' + region.upper() if region else '') | Evidence: print(f'{c} -> {norm(c)}')

🛠️ Reader to Writer Bridge

1. Read: explain normalize hreflang code casing in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 358.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 360 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c} -> {norm(c)}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `raw = ['EN-us', 'Fr-fr', 'de-DE', 'ES']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of normalize hreflang code casing and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Partitioning on `-` and applying `lower()` to the language and `upper()` to the region rebuilds each code in its conventional canonical casing.

```
raw = ['EN-us', 'Fr-fr', 'de-DE', 'ES']
def norm(c):
    lang, _, region = c.partition('-')
    return lang.lower() + ('-' + region.upper() if region else '')
for c in raw:
    print(f'{c} -> {norm(c)}')
```

```
output
```

```
EN-us -> en-US
Fr-fr -> fr-FR
de-DE -> de-DE
ES -> es
```

How it works:

1. A list named raw holds codes in inconsistent casing, including the region-less 'ES'.
2. norm uses partition('-') to split, then returns lang.lower() plus, only if a region exists, '-' and region.upper().
3. The loop prints each original code mapped to its normalized form, so 'EN-us' becomes 'en-US'.

Try it:

1. Strip surrounding whitespace before normalizing each code.
2. Preserve the special 'x-default' token untouched through norm.
3. Return None for codes that cannot be parsed into a valid shape.
4. Add `ZH-hant` and decide how to canonicalize a script subtag.

⚠ Common Mistakes

- Lowercasing the entire code turns `de-DE` into `de-de` and loses the region convention.
- Forgetting the region-less case would append a stray `-` to a bare language like `es`.
- Title-casing the language (`En`) is neither convention nor valid for matching.

Skills gained partition normalization functions casing

→ **Where You'll Use This:** Before deduplicating or comparing hreflang codes scraped from many pages, normalizing casing prevents 'en-US' and 'EN-us' from being treated as different locales in your audit data.

Mental Model

Normalization is alphabetizing a contacts list by a fixed rule so that 'McLeod' and 'Mcleod' file together instead of drifting apart.

Project 360 Map a Locale to Its Currency

The Idea Behind It

A localized storefront should show prices in each region's own currency. This project keeps a lookup table mapping serviced locales to currency codes, then resolves several locales through it, using dict.get with a default so an unconfigured locale like 'en-CA' is flagged UNMAPPED rather than crashing or silently defaulting.

What You'll Build & Learn

Map each serviced locale to its currency and flag any locale that has not been configured.

- Locales drive storefront currency
- dict.get with a default avoids errors
- Flagging unconfigured locales

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 359: normalize hreflang code casing. Project 361 changes the surface problem while preserving the underlying method.

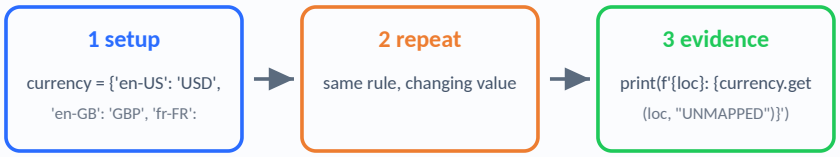
progress check

Reading target: map a locale to its currency is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `currency = {'en-US': 'USD', 'en-GB': 'GBP', 'fr-FR': 'EUR', 'ja-JP': 'JPY'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{loc}: {currency.get(loc, "UNMAPPED")}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`currency = {'en-US': 'USD', 'en-GB': 'GBP', 'fr-FR': 'EUR', 'ja-JP': 'JPY'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

Move	Code to inspect	Why it matters
2. Apply the rule	`for loc in ['en-US', 'fr-FR', 'en-CA']:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{loc}: {currency.get(loc, "UNMAPPED")}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for loc in ['en-US', 'fr-FR', 'en-CA']:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Map a Locale to Its Currency: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: currency, en, US, GB, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: currency = {'en-US': 'USD', 'en-GB': 'GBP', 'fr-FR': 'EUR', 'ja-JP': 'J... | Core: for loc in ['en-US', 'fr-FR', 'en-CA']: | Evidence: print(f'{loc}: {currency.get(loc, "UNMAPPED")}')

Reader to Writer Bridge

1. Read: explain map a locale to its currency in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 359.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 361 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{loc}: {currency.get(loc, "UNMAPPED")}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `currency = {'en-US': 'USD', 'en-GB': 'GBP', 'fr-FR': 'EUR', 'ja-JP': 'JPY'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of map a locale to its currency and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A dict from locale to ISO 4217 currency code, queried with `.get` and a sentinel, reports the currency or surfaces an unmapped locale.

```
currency = {'en-US': 'USD', 'en-GB': 'GBP', 'fr-FR': 'EUR', 'ja-JP': 'JPY'}
for loc in ['en-US', 'fr-FR', 'en-CA']:
    print(f'{loc}: {currency.get(loc, "UNMAPPED")}')
```

```
output
en-US: USD
fr-FR: EUR
en-CA: UNMAPPED
```

How it works:

1. A dict named currency maps four locales such as 'en-US' to currency codes like 'USD'.
2. The loop iterates a list of locales to resolve, including 'en-CA' which is absent from the table.
3. currency.get(loc, 'UNMAPPED') returns the currency or the fallback string, which prints for the missing Canadian locale.

Try it:

1. Add tax rate and payment methods alongside currency per locale.
2. Raise an error instead of printing UNMAPPED to fail a build.
3. Group locales by shared currency such as several EUR markets.

4. Add `en-CA -> CAD` and watch it resolve.

⚠ Common Mistakes

- Mapping on language alone (`en`) ignores that `en-US` and `en-GB` use different currencies.
- Indexing with `currency[loc]` raises a KeyError for any locale you forgot to configure.
- Hard-coding one currency per language breaks every multi-region language.

Skills gained dicts | lookup tables | get default | localization

→ Where You'll Use This: International commerce sites tie locale to currency, tax, and payment options; this lookup pattern is the basis for catching a market you launched without setting up its currency.

⚡ Mental Model

The lookup table is a currency exchange board at the airport: if your country is not listed, the clerk shrugs instead of guessing.

Project 361 Route a Visitor to a Default Locale

💡 The Idea Behind It

When none of a visitor's preferred locales are available you serve a default. This project keeps a set of served locales and a fixed default, then defines a route function that walks the visitor's ordered preferences and returns the first one that is served, falling back to the default when nothing matches, modeling server-side locale routing.

What You'll Build & Learn

Route a visitor to the first of their preferred locales that you serve, defaulting when none match.

- Honor ordered visitor preferences
- Fall back to a default locale
- First served preference wins

🧭 Skill Roadmap

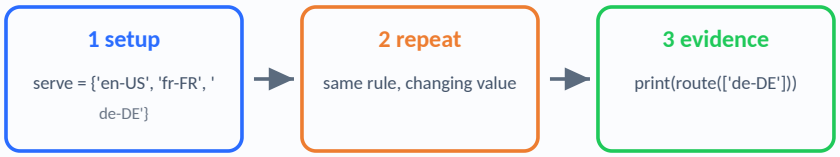
Skill step: pushes the same idea into a less automatic situation. It builds on Project 360: map a locale to its currency. Project 362 changes the surface problem while preserving the underlying method. *progress check*

Reading target: route a visitor to a default locale is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `serve = {'en-US', 'fr-FR', 'de-DE'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(route(['de-DE']))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`serve = {'en-US', 'fr-FR', 'de-DE'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if pref in serve:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(route(['de-DE']))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if pref in serve:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Route a Visitor to a Default Locale: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: serve, en, US, fr, FR. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: serve = {'en-US', 'fr-FR', 'de-DE'} | Core: if pref in serve: | Evidence: print(route(['de-DE']))

🛠️ Reader to Writer Bridge

1. Read: explain route a visitor to a default locale in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 360.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 362 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(route(['de-DE']))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `serve = {'en-US', 'fr-FR', 'de-DE'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of route a visitor to a default locale and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Returning the first preference present in the served set, or the default when none match, implements priority-ordered locale negotiation.

```
serve = {'en-US', 'fr-FR', 'de-DE'}
default = 'en-US'
def route(accept):
    for pref in accept:
        if pref in serve:
            return pref
    return default
print(route(['fr-FR', 'en-US']))
print(route(['it-IT', 'es-ES']))
print(route(['de-DE']))
```

```
output
fr-FR
en-US
de-DE
```

How it works:

1. A set named serve lists the three locales the site offers and default holds 'en-US'.
2. route loops over the visitor's accept list and returns the first pref found in serve.
3. Three calls show a served preference winning, an all-unavailable list dropping to the default, and a single served locale being chosen.

Try it:

1. Fall back to bare language before the global default.
2. Return the matched locale's URL rather than its code.
3. Log when the hard default was used so you can measure mismatches.
4. Add `es-ES` to the served set and re-run the Spanish-leaning request.

⚠ Common Mistakes

- Returning the default before exhausting the visitor's preference list ignores valid matches.
- Treating preferences as an unordered set loses the visitor's priority order.
- Using a default that is not in the served set can route users to a missing page.

Skills gained functions routing preferences defaults

→ Where You'll Use This: Geo and language routing at the edge or in middleware uses this priority-then-default logic; getting it right keeps visitors on a relevant page instead of a jarring wrong-language landing.

Mental Model

Routing is a host seating guests by request: first choice if free, second if not, and a standard table when nothing they asked for is open.

Project 362 Pick a Locale from Accept-Language

The Idea Behind It

Browsers send an Accept-Language header listing locales with quality weights like 'en-US;q=1.0'. This project parses that header by splitting on commas, partitioning each part on ';q=' to separate the code from its weight, then choosing the highest-weighted locale with max over (weight, code) tuples, showing how a server reads a visitor's true priority.

What You'll Build & Learn

Parse an Accept-Language header with quality weights and pick the visitor's highest-priority locale.

- Accept-Language carries q quality weights
- Partitioning a header field on ';q='
- max over tuples picks the top weight

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 361: route a visitor to a default locale. Project 363 changes the surface problem while preserving the underlying method.

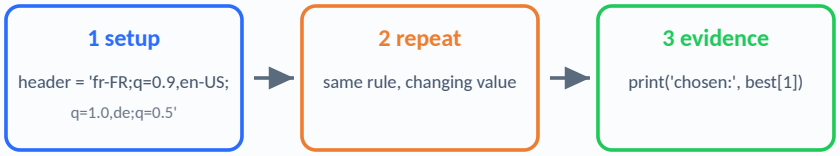
progress check

Reading target: pick a locale from accept-language is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `header = 'fr-FR;q=0.9,en-US;q=1.0,de;q=0.5'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('chosen:', best[1])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`header = 'fr-FR;q=0.9,en-US;q=1.0,de;q=0.5'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`parsed.append((float(q) if q else 1.0, code))`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('chosen:', best[1])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `parsed.append((float(q) if q else 1.0, code))`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Pick a Locale from Accept-Language: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: header, fr, FR, en, US. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: header = 'fr-FR;q=0.9,en-US;q=1.0,de;q=0.5' | Core: parsed.append *float(q) if q else 1.0, code* | Evidence: print('chosen:', best[1])

Reader to Writer Bridge

1. Read: explain pick a locale from accept-language in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 361.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 363 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print('chosen:', best[1])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `header = 'fr-FR;q=0.9,en-US;q=1.0,de;q=0.5'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of pick a locale from accept-language and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Splitting each part on `;q=` separates code from weight, defaulting to 1.0 when absent, and `max` over `(weight, code)` tuples selects the strongest preference.

```
header = 'fr-FR;q=0.9,en-US;q=1.0,de;q=0.5'
parsed = []
for part in header.split(','):
    code, _, q = part.partition(';q=')
    parsed.append((float(q) if q else 1.0, code))
best = max(parsed)
```

```
print('weights:', sorted(parsed, reverse=True))
print('chosen:', best[1])
```

```
output
weights: [(1.0, 'en-US'), (0.9, 'fr-FR'), (0.5, 'de')]
chosen: en-US
```

How it works:

1. The header string lists three locales each with a ;q= weight, comma separated.
2. Splitting on commas and partitioning each part on ';q=' yields the code and weight, stored as (float(q) if q else 1.0, code) tuples.
3. max(parsed) returns the tuple with the highest weight, and the prints show the sorted weights plus the chosen 'en-US'.

Try it:

1. Handle a malformed q-value gracefully instead of crashing.
2. Break ties by preferring a more specific locale over a bare language.
3. Filter to only locales the site actually serves before choosing.
4. Give `de` a weight of 1.0 and watch the tie-break behaviour.

⚠ Common Mistakes

- Treating a missing `;q=` as weight 0 instead of the spec's default of 1.0.
- Comparing weights as strings makes `0.9` outrank `1.0` lexically.
- Forgetting that whitespace can surround entries in a real header and break the split.

Skills gained parsing | accept-language | tuples | max

→ Where You'll Use This: Servers that personalize language without a stored preference rely on Accept-Language; parsing its q-values correctly is what decides the locale a first-time visitor sees.

Mental Model

The q-weights are a ranked-choice ballot: the visitor lists languages in order of preference and the server tallies the winner.

Project 363 Validate the x-default Target

The Idea Behind It

The x-default URL should usually be a neutral language selector or root page, distinct from any one locale's URL. This project pulls the x-default target out of a cluster, gathers the set of all the other locale URLs, and

checks that the x-default is not among them, catching a fallback misconfigured to duplicate a specific market's page.

What You'll Build & Learn

Verify the x-default target is distinct from every specific locale URL in the cluster.

- x-default should be neutral
- It must differ from locale URLs
- Set membership tests distinctness

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 362: pick a locale from accept-language. Project 364 changes the surface problem while preserving the underlying method.

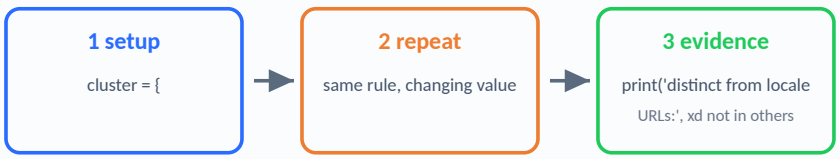
progress check

Reading target: validate the x-default target is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `cluster = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('distinct from locale URLs:', xd not in others)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`cluster = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`others = {u for c, u in cluster.items() if c != 'x-default'}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('distinct from locale URLs:', xd not in others)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `others = {u for c, u in cluster.items() if c != 'x-default'}`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Validate the x-default Target: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: cluster, en, US, https, com. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: cluster = { | Core: others = {u for c, u in cluster.items() if c != 'x-default'} | Evidence: print('distinct from locale URLs:', xd not in others)

🛠️ Reader to Writer Bridge

1. Read: explain validate the x-default target in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 362.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 364 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('distinct from locale URLs:', xd not in others)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `cluster = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of validate the x-default target and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Collecting the non-default locale URLs into a set and testing that the `x-default` target is not among them confirms the fallback is its own neutral page.

```
cluster = {
    'en-US': 'https://s.com/us/',
    'en-GB': 'https://s.com/gb/',
    'x-default': 'https://s.com/',
}
xd = cluster.get('x-default')
others = {u for c, u in cluster.items() if c != 'x-default'}
print('x-default ->', xd)
print('distinct from locale URLs:', xd not in others)
```

```
output
x-default -> https://s.com/
distinct from locale URLs: True
```

How it works:

1. A cluster dict maps two English locales plus 'x-default' to their URLs, with x-default pointing at the bare root.

2. cluster.get('x-default') extracts the fallback URL, and a set comprehension gathers every other locale's URL.
3. xd not in others prints True, confirming the x-default does not collide with a specific locale page.

Try it:

1. Warn if x-default points at a locale-specific subpath like /us/.
2. Allow the x-default to equal the root only if a selector exists there.
3. Run the check across many clusters and total the violations.
4. Point `x-default` at the US URL and watch the distinctness check fail.

⚠ Common Mistakes

- Pointing `x-default` at a specific locale (e.g. the US page) defeats its neutral purpose.
- Comparing against locale codes rather than their URLs gives a meaningless check.
- Assuming any present `x-default` is correct without inspecting where it points.

Skills gained sets | x-default | membership | validation

→ Where You'll Use This: A misconfigured x-default that simply repeats the US page tells Google to send unmatched visitors into a market-specific experience; this distinctness check guards against that subtle fallback error.

⚡ Mental Model

x-default should be the neutral reception desk, not someone's private office; sending lost visitors into a single office is a misroute.

Project 364 Group Locale Pages by Language

💡 The Idea Behind It

Reporting locales grouped by language shows regional coverage at a glance: which languages you serve and in how many regions each. This project uses a defaultdict of lists to bucket every code under its language subtag by splitting on the hyphen, then prints each language with its sorted regional variants, the inverse of flattening a matrix.

What You'll Build & Learn

Group locale codes by their language so you can read regional coverage per language at a glance.

- Grouping reveals regional coverage
- defaultdict(list) buckets by key
- Splitting code to find its language

Skill Roadmap

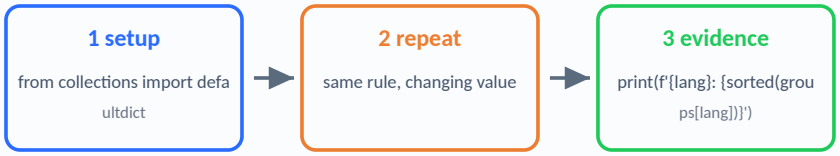
Skill step: pushes the same idea into a less automatic situation. It builds on Project 363: validate the x-default target. Project 365 changes the surface problem while preserving the underlying method. *progress check*

Reading target: group locale pages by language is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import defaultdict` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{lang}: {sorted(groups[lang])}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import defaultdict`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`for c in codes:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{lang}: {sorted(groups[lang])}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for c in codes:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Group Locale Pages by Language: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: collections, defaultdict, codes, en, US. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import defaultdict | Core: for c in codes: | Evidence: print(f'{lang}: {sorted(groups[lang])}')

🛠️ Reader to Writer Bridge

1. Read: explain group locale pages by language in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 363.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 365 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{lang}: {sorted(groups[lang])}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from collections import defaultdict` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of group locale pages by language and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A `defaultdict(list)` keyed by the language subtag collects each region under its language, and sorting both the keys and the grouped lists keeps the output deterministic.

```
from collections import defaultdict
codes = ['en-US', 'en-GB', 'fr-FR', 'fr-CA', 'de-DE']
groups = defaultdict(list)
for c in codes:
    groups[c.split('-')[0]].append(c)
for lang in sorted(groups):
    print(f'{lang}: {sorted(groups[lang])}')
```

```
output
de: ['de-DE']
en: ['en-GB', 'en-US']
fr: ['fr-CA', 'fr-FR']
```

How it works:

1. A list named codes holds five locale variants across English, French, and German.
2. A defaultdict(list) named groups appends each code under groups[c.split('-')[0]], its language part.
3. Iterating sorted(groups) prints each language followed by its sorted list of regional variants.

Try it:

1. Append the count of regions next to each language.
2. Sort languages by how many regions they cover.
3. Handle a bare language code with no region in the grouping.
4. Add `es-ES` and `es-MX` to see a new language group form.

⚠ Common Mistakes

- Grouping by the full code keeps `en-US` and `en-GB` apart and defeats the purpose.
- Using a plain dict and indexing a missing key raises instead of starting a new group.

- Forgetting to sort the per-language lists makes the regions print in arbitrary order.

Skills gained defaultdict | grouping | string split | reporting

→ Where You'll Use This: Coverage reports for stakeholders read better grouped by language than as a flat list; this grouping shows at once that English serves two regions while German serves only one.

Mental Model

Grouping by language is sorting a bookshelf by author, then seeing at a glance who wrote one book and who wrote ten.

Project 365 Flag a Duplicate Region per Language

The Idea Behind It

Declaring the same locale twice, such as two 'en-US' tags, is a duplication bug. This project tallies a list of locale codes with a Counter, then prints each code alongside its occurrence count, appending a DUPLICATE flag to any code that appears more than once so repeated region declarations stand out.

What You'll Build & Learn

Tally locale codes and flag any that appear more than once as duplicate region declarations.

- Repeating a locale is a bug
- Counter gives per-code tallies
- Flagging counts above one

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 364: group locale pages by language. Project 366 changes the surface problem while preserving the underlying method. *progress check*

Reading target: flag a duplicate region per language is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

- Evidence: `print(f'{c}: x{counts[c]}{flag}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`flag = ' DUPLICATE' if counts[c] > 1 else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{c}: x{counts[c]}{flag}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = ' DUPLICATE' if counts[c] > 1 else ''`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag a Duplicate Region per Language: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: collections, Counter, codes, en, US. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: flag = ' DUPLICATE' if counts[c] > 1 else '' | Evidence: print(f'{c}: x{counts[c]}{flag}')

🛠️ Reader to Writer Bridge

1. Read: explain flag a duplicate region per language in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 364.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 366 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{c}: x{counts[c]} {flag}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag a duplicate region per language and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `Counter` over the codes gives each one's occurrence count, and printing a flag whenever the count exceeds one surfaces the duplicates in sorted order.

```
from collections import Counter
codes = ['en-US', 'en-GB', 'en-US', 'fr-FR']
counts = Counter(codes)
for c in sorted(counts):
```

```
    flag = ' DUPLICATE' if counts[c] > 1 else ''
    print(f'{c}: x{counts[c]}{flag}')
```

```
output
en-GB: x1
en-US: x2 DUPLICATE
fr-FR: x1
```

How it works:

1. A list named codes has 'en-US' twice among the four entries.
2. Counter(codes) builds a tally of how often each code appears.
3. Iterating sorted(counts) prints each code with 'x' and its count, adding ' DUPLICATE' when the count exceeds one.

Try it:

1. Exit with an error code if any duplicate is found.
2. Report only the duplicates instead of every code.
3. Show which language has the most duplicated regions.
4. Add a third `en-US` and watch the multiplier rise to x3.

⚠ Common Mistakes

- Deduping with a set first erases the very repetition you want to report.
- Flagging codes that merely share a language (`en-US`, `en-GB`) confuses regions with duplicates.
- Iterating the Counter unsorted makes the report order vary run to run.

Skills gained Counter duplicates collections auditing

→ Where You'll Use This: A page that injects the same region tag twice often comes from a looping template; the per-code tally turns that defect into a clear, sorted line item in your QA output.

⚡ Mental Model

The Counter is a turnstile clicker: it just counts each badge as it passes and lights up when the same one comes through twice.

Project 366 Generate a Reciprocal Hreflang Set

💡 The Idea Behind It

Given one cluster, the correct hreflang block is identical on every page: each lists all locales including itself. This project iterates the cluster once per page and, for each, prints the full set of hreflang declarations, generating a reciprocal block programmatically so every page ends up with the same complete, mutually-linking tags.

What You'll Build & Learn

Generate the identical, fully reciprocal hreflang block that every page in a cluster should carry.

- Every page shares one identical block
- Generation guarantees reciprocity
- Nested iteration over a cluster

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 365: flag a duplicate region per language. Project 367 changes the surface problem while preserving the underlying method.

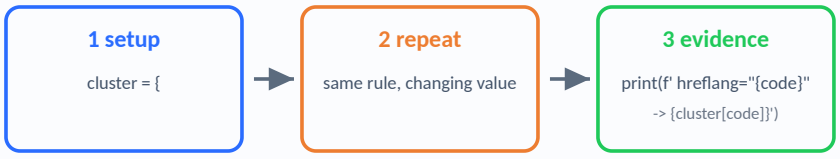
progress check

Reading target: generate a reciprocal hreflang set is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `cluster = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f' hreflang="{code}" -> {cluster[code]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`cluster = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`for page_code in sorted(cluster):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f' hreflang="{code}" -> {cluster[code]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for page_code in sorted(cluster):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Generate a Reciprocal Hreflang Set: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: cluster, en, https, com, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: cluster = { | Core: for page_code in sorted(cluster): | Evidence: print(f' hreflang="{code}" -> {cluster[code]}')

🛠️ Reader to Writer Bridge

1. Read: explain generate a reciprocal hreflang set in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 365.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 367 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f' hreflang="{code}" -> {cluster[code]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `cluster = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of generate a reciprocal hreflang set and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For each page in the cluster, emitting a tag for every locale (sorted) produces the same complete, self-including block everywhere, which is exactly what reciprocity requires.

```
cluster = {
    'en': 'https://s.com/en/',
    'fr': 'https://s.com/fr/',
    'de': 'https://s.com/de/',
}
for page_code in sorted(cluster):
    print(f'On {cluster[page_code]}:')
    for code in sorted(cluster):
        print(f'  hreflang="{code}" -> {cluster[code]}')
```

```
output
On https://s.com/de/:
  hreflang="de" -> https://s.com/de/
  hreflang="en" -> https://s.com/en/
  hreflang="fr" -> https://s.com/fr/
On https://s.com/en/:
  hreflang="de" -> https://s.com/de/
  hreflang="en" -> https://s.com/en/
  hreflang="fr" -> https://s.com/fr/
On https://s.com/fr/:
```

```
  hreflang="de" -> https://s.com/de/
  hreflang="en" -> https://s.com/en/
  hreflang="fr" -> https://s.com/fr/
```

How it works:

1. A cluster dict maps three language codes to their absolute URLs.
2. The outer loop over sorted(cluster) prints an 'On URL:' header for each page.
3. The inner loop over sorted(cluster) prints every hreflang code and its URL, so each page emits the same complete reciprocal block.

Try it:

1. Append an x-default line to every generated block.
2. Output real link rel='alternate' HTML tags instead of arrows.
3. Emit the equivalent XML sitemap hreflang entries instead.
4. Add an `x-default` entry and watch every block gain it.

⚠ Common Mistakes

- Omitting a page's own locale from its block breaks the self-reference rule.
- Generating a different subset per page reintroduces the reciprocity bugs you meant to avoid.
- Iterating the cluster unsorted makes the generated tags print in unstable order.

Skills gained nested loops | generation | reciprocity | dicts

→ Where You'll Use This: Generating hreflang from a single source of truth, rather than hand-editing per page, is the practical way to guarantee reciprocity and self-references stay perfect as locales are added.

⚡ Mental Model

It is photocopying one master page for everyone: because the original is correct, every copy is automatically correct and consistent.

Project 367 Audit a Cluster for Required Tags

💡 The Idea Behind It

A cluster passes only if it self-references, declares an x-default, and lists at least two locales. This project bundles those three conditions into a checks dict of named booleans, evaluating each against the cluster's codes, then prints a PASS or FAIL line per rule to deliver a quick compliance verdict in one place.

What You'll Build & Learn

Run three required-tag checks on a cluster at once and report a pass or fail for each.

- Three rules define a passing cluster
- A dict of named boolean checks
- Reporting each rule's verdict

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 366: generate a reciprocal hreflang set. Project 368 changes the surface problem while preserving the underlying method.

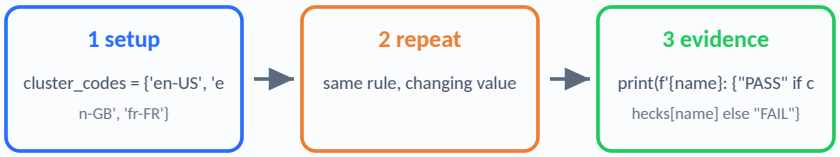
progress check

Reading target: audit a cluster for required tags is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `cluster_codes = {'en-US', 'en-GB', 'fr-FR'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print(f'{name}: {"PASS" if checks[name] else "FAIL"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`cluster_codes = {'en-US', 'en-GB', 'fr-FR'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.

Move	Code to inspect	Why it matters
2. Apply the rule	`print(f'{name}: {"PASS" if checks[name] else "FAIL"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{name}: {"PASS" if checks[name] else "FAIL"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{name}: {"PASS" if checks[name] else "FAIL"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Audit a Cluster for Required Tags: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: cluster_codes, en, US, GB, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: cluster_codes = {'en-US', 'en-GB', 'fr-FR'} | Core: print(f'{name}: {"PASS" if checks[name] else "FAIL"}') | Evidence: print(f'{name}: {"PASS" if checks[name] else "FAIL"}')

Reader to Writer Bridge

1. Read: explain audit a cluster for required tags in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 366.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 368 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{name}: {"PASS" if checks[name] else "FAIL"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `cluster_codes = {'en-US', 'en-GB', 'fr-FR'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of audit a cluster for required tags and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Encoding each requirement as a boolean in a dict lets you print a labelled PASS/FAIL line per rule, making a partial failure (here, the missing x-default) obvious.

```
cluster_codes = {'en-US', 'en-GB', 'fr-FR'}
page_locale = 'en-US'
checks = {
    'self present': page_locale in cluster_codes,
    'x-default present': 'x-default' in cluster_codes,
    'at least 2 locales': len(cluster_codes) >= 2,
}
for name in sorted(checks):
    print(f'{name}: {"PASS" if checks[name] else "FAIL"}')
```

```
output
at least 2 locales: PASS
self present: PASS
x-default present: FAIL
```

How it works:

1. cluster_codes is a set and page_locale names the page's own locale, here 'en-US'.
2. A checks dict maps three rule names to booleans: self present, x-default present, and at least two locales.
3. Iterating sorted(checks) prints each rule with PASS or FAIL, exposing that x-default is missing.

Try it:

1. Add a rule requiring all codes to be well-formed.

2. Compute an overall PASS only when every check passes.
3. Raise the locale minimum to three and observe the change.
4. Add `x-default` to the code set and watch that line flip to PASS.

⚠ Common Mistakes

- Combining the checks into one boolean hides which specific rule failed.
- Counting `x-default` toward the 'at least two locales' rule inflates the locale count.
- Iterating the checks dict unsorted makes the report order vary.

Skills gained dicts | boolean checks | compliance | sets

→ Where You'll Use This: A compact pass/fail panel per cluster is what a dashboard surfaces to non-technical stakeholders; bundling the named rules keeps the report readable and easy to extend with new criteria.

Mental Model

The checks dict is a pre-flight checklist: each item gets its own tick, and the plane only leaves when they all read green.

Project 368 Check Alternates Share a Scheme

The Idea Behind It

Alternates may live on subdomains or ccTLDs, but they should at least share a single scheme such as all https. This project extracts the scheme and host from each alternate URL by splitting on '://' and the first slash, then reports the distinct schemes and hosts and confirms whether exactly one scheme is in use, revealing the cluster's URL topology.

What You'll Build & Learn

Extract schemes and hosts from alternate URLs and confirm the whole cluster shares one scheme.

- Alternates can span subdomains or ccTLDs
- They should share a single scheme
- Splitting URLs into scheme and host

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 367: audit a cluster for required tags. Project 369 changes the surface problem while preserving the underlying method.

progress check

Reading target: check alternates share a scheme is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `alts = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('single scheme:', len(schemes) == 1)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`alts = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`schemes = {u.split('://', 1)[0] for u in alts.values()}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('single scheme:', len(schemes) == 1)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `schemes = {u.split('://', 1)[0] for u in alts.values()}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Alternates Share a Scheme: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: alts, en, https, com, fr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: alts = { | Core: schemes = {u.split('://', 1)[0] for u in alts.values()} | Evidence: print('single scheme:', len(schemes) == 1)

🛠️ Reader to Writer Bridge

1. Read: explain check alternates share a scheme in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 367.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 369 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('single scheme:', len(schemes) == 1)`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `alts = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check alternates share a scheme and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Splitting each URL on `://` isolates the scheme, and splitting the remainder on the first `/` isolates the host, so set sizes reveal whether the cluster mixes schemes or hosts.

```
alts = {
    'en': 'https://s.com/en/',
    'fr': 'https://s.com/fr/',
    'de': 'https://de.s.com/',
}
schemes = {u.split('://', 1)[0] for u in alts.values()}
hosts = {u.split('://', 1)[1].split('/', 1)[0] for u in alts.values()}
print('schemes:', sorted(schemes))
print('hosts:', sorted(hosts))
print('single scheme:', len(schemes) == 1)
```

```
output
schemes: ['https']
hosts: ['de.s.com', 's.com']
single scheme: True
```

How it works:

1. An alts dict maps locales to URLs, one on the de.s.com subdomain.
2. A set comprehension splits each URL on '://' to collect schemes, and another splits the remainder on the first slash to collect hosts.
3. The prints show the sorted schemes and hosts, and len(schemes) == 1 confirms a single shared scheme.

Try it:

1. Flag any http URL as a downgrade from https.
2. Report whether all hosts share a common registrable domain.
3. Use urllib.parse for robust scheme and host extraction.
4. Change one alternate to a relative path and watch the scheme split fail to find https.

⚠ Common Mistakes

- Assuming differing hosts is an error when subdomain or ccTLD strategies legitimately differ.
- Splitting on `/` without first removing the scheme captures the wrong host segment.

- Forgetting to sort the scheme and host sets makes the output nondeterministic.

Skills gained `string split` `sets` `url parsing` `schemes`

→ **Where You'll Use This:** International sites legitimately mix subdomains and country domains; checking that they at least share https avoids the mixed-scheme inconsistency that erodes trust and complicates canonical handling.

Mental Model

The scheme is the language everyone speaks at a multinational meeting: the offices differ, but the conversation should stay in one tongue.

Project 369 Score an Hreflang Cluster's Health

The Idea Behind It

Rolling several checks into a single 0-to-100 score gives stakeholders a one-glance health metric. This project weights three rules, valid code shapes worth forty points, x-default presence worth thirty, and all-https alternates worth thirty, multiplying each all-pass boolean by its weight and summing them into a composite hreflang health score.

What You'll Build & Learn

Combine validity, x-default, and https checks into a weighted 0-100 hreflang health score.

- Weighted rules form a composite score
- Booleans multiply by point values
- A single metric for stakeholders

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 368: check alternates share a scheme. Project 370 changes the surface problem while preserving the underlying method.

progress check

Reading target: score an hreflang cluster's health is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

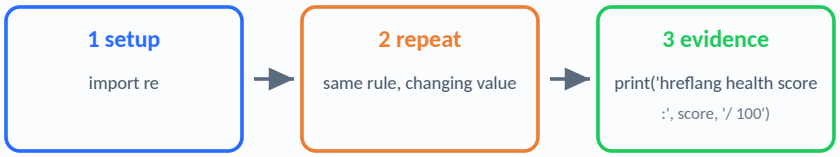
Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('hreflang health score:', score, '/ 100')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`score += all(c == 'x-default' or pat.match(c) for c in cluster) * 40`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('hreflang health score:', score, '/ 100')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `score += all(c == 'x-default' or pat.match(c) for c in cluster) * 40`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score an Hreflang Cluster's Health: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: re, cluster, en, US, https. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: score += all(c == 'x-default' or pat.match(c) for c in cluster) * 40 | Evidence: print('hreflang health score:', score, '/ 100')

🛠️ Reader to Writer Bridge

1. Read: explain score an hreflang cluster's health in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 368.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 370 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('hreflang health score:', score, '/ 100')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of score an hreflang cluster's health and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Multiplying each boolean check by its point weight and summing yields a deterministic score; valid codes, an x-default, and absolute URLs here total a perfect 100.

```
import re
cluster = {'en-US': 'https://s.com/en/', 'fr-FR': 'https://s.com/fr/', 'x-
default': 'https://s.com/'}
pat = re.compile(r'^[a-z]{2}(-[A-Z]{2})?$')
score = 0
score += all(c == 'x-default' or pat.match(c) for c in cluster) * 40
score += ('x-default' in cluster) * 30
score += all(u.startswith('https://') for u in cluster.values()) * 30
print('hreflang health score:', score, '/ 100')
```

```
output
hreflang health score: 100 / 100
```

How it works:

1. A cluster dict and the shape regex pat are defined, and score starts at zero.
2. Each rule's all(...) boolean is multiplied by its weight: 40 for valid codes (treating 'x-default' as exempt), 30 for x-default present, and 30 for all-https.
3. Summing the three weighted terms yields the final score, printed here as 100 out of 100.

Try it:

1. Add a reciprocity check worth its own slice of the score.
2. Return a breakdown of which rules earned which points.
3. Make the weights configurable so teams can re-prioritize.
4. Drop the `x-default` entry and watch the score fall to 70.

⚠ Common Mistakes

- Forgetting to exempt `x-default` from the code regex zeroes the validity slice.
- Letting the weights sum to something other than 100 makes the score hard to read.
- Using `and` results without multiplying by weights collapses the score to a bare boolean.

Skills gained scoring | weighting | regex | boolean arithmetic

→ Where You'll Use This: A single health score lets you trend hreflang quality over time and across templates in a dashboard, turning a pile of individual checks into one number leadership can track.

Mental Model

The score is a credit rating: many underlying factors collapse into one number that everyone can compare at a glance.

Project 370 Run a Full Hreflang Audit Report

The Idea Behind It

This capstone ties the chapter together across a real page-to-alternates map. In one pass it validates each code against the shape regex while skipping x-default, checks every alternate target for a return tag, and confirms x-default appears on every page, then prints the count of pages, invalid codes, broken return tags, and overall x-default coverage as the skeleton of an international SEO audit.

What You'll Build & Learn

Run validity, reciprocity, and x-default coverage checks together over a page map as one combined audit.

- Combining several audits in one pass
- Skipping x-default during code validation
- Reporting a multi-metric summary

Skill Roadmap

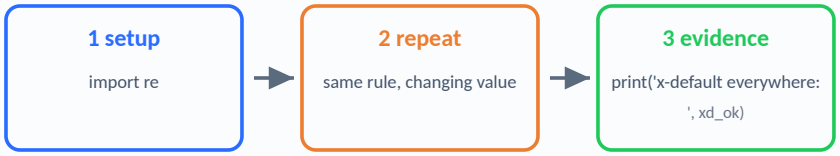
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 369: score an hreflang cluster's health. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run a full hreflang audit report is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
- Evidence: `print('x-default everywhere:', xd_ok)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
2. Apply the rule	`if code == 'x-default':`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('x-default everywhere:', xd_ok)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: International SEO and hreflang becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if code == 'x-default':`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Run a Full Hreflang Audit Report: connect the surface task to the chapter skill, validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay.
Names	Watch these names first: re, pages, https, com, en. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if code == 'x-default': | Evidence: print('x-default everywhere:', xd_ok)

🛠️ Reader to Writer Bridge

1. Read: explain run a full hreflang audit report in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 369.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('x-default everywhere:', xd_ok)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run a full hreflang audit report and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

One walk over every page's alternates accumulates invalid codes and broken return tags while a separate check confirms x-default everywhere, summarising cluster health in four lines.

```
import re
pages = {
    'https://s.com/en/': {'en': 'https://s.com/en/', 'fr': 'https://s.com/fr/',
'x-default': 'https://s.com/'},
    'https://s.com/fr/': {'en': 'https://s.com/en/', 'fr': 'https://s.com/fr/',
'x-default': 'https://s.com/'},
}
pat = re.compile(r'^[a-z]{2}(-[A-Z]{2})?$')
```

```
bad_codes, no_return = [], []
for page, alts in pages.items():
    for code, target in alts.items():
        if code == 'x-default':
            continue
        if not pat.match(code):
            bad_codes.append(code)
        if page not in pages.get(target, {}).values():
            no_return.append((page, target))
xd_ok = all('x-default' in a for a in pages.values())
print('pages audited:', len(pages))
print('invalid codes:', sorted(set(bad_codes)))
print('broken return tags:', len(no_return))
print('x-default everywhere:', xd_ok)
```

```
output
pages audited: 2
invalid codes: []
broken return tags: 0
x-default everywhere: True
```

How it works:

1. A pages dict maps two page URLs to their alternate dicts including an 'x-default' entry each.
2. Nested loops skip the x-default code, append any code that fails pat.match to bad_codes, and append a pair to no_return when the target does not link back.
3. xd_ok checks 'x-default' appears in every page's alternates, and four prints summarize pages audited, invalid codes, broken return tags, and x-default coverage.

Try it:

1. Add duplicate and conflict detection per page to the same pass.
2. Emit the report as JSON for a monitoring pipeline to ingest.
3. Include orphan-alternate detection against a set of real URLs.
4. Break one return tag and watch the broken count rise.

⚠ Common Mistakes

- Running the code regex on `x-default` flags the fallback as an invalid locale.
- Counting return tags against the x-default URL (which need not reciprocate) inflates the broken count.
- Reporting raw lists without `set` dedupe repeats the same invalid code many times.

Skills gained regex | reciprocity | x-default | auditing

→ Where You'll Use This: This is the shape of a production hreflang auditor: feed it a crawl's page-to-alternates map and it returns the headline numbers an SEO team needs to triage international tagging issues.

Mental Model

The capstone is a full health screening: one appointment runs every test and hands back a single chart covering all the vital signs.

Chapter Progress Checkpoint

What this chapter added: after 32 mini-projects, International SEO and hreflang should feel like a reusable Python move: validate hreflang matrices and return-tag reciprocity, locale/region codes and canonical interplay. *chapter checkpoint*

1 Read
Open any project from Projects 339-370 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.
transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 11 (Off-Page, Links and Authority) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

8 questions · 4 categories · graded

2 Recall | 3 Meaning | 1 Structure | 2 Transfer

10.1.Q1 Recall MEDIUM — Turn back to Project 363 (Validate the x-default Target). Without running it, what exactly does it print?

10.1.Q2 Meaning MEDIUM — Turn back to Project 351 (Count Distinct Locales in a Cluster). Without running it, what exactly does it print?

10.1.Q3 Structure MEDIUM — Turn back to Project 342 (Detect a Malformed Hreflang Code). Without running it, what exactly does it print?

10.1.Q4 Transfer MEDIUM — Turn back to Project 362 (Pick a Locale from Accept-Language). Without running it, what exactly does it print?

10.1.Q5 Recall MEDIUM In the hreflang code **`fr-CA`**, which part is the language and which is the region, and how is each cased?

- A `fr` region uppercase, CA language lowercase
- B `fr` language lowercase (ISO 639-1), CA region uppercase (ISO 3166-1 alpha-2)
- C Both are languages
- D CA is the language because it comes first in importance

10.1.Q6 Meaning MEDIUM Page A declares page B as an **`fr`** alternate, but page B does not list page A anywhere in its hreflang block. What happens?

- A Google still honours the A→B link
- B The relationship is ignored because the return tag is missing (reciprocity is broken)
- C Page B is deleted from the index
- D Nothing — return tags are optional

10.1.Q7 Meaning MEDIUM Your French page (**`/fr/`**) sets its canonical to the English page (**`/en/`**). What is the likely effect on the French page and its hreflang?

- A French ranks better in France
- B French is treated as a duplicate of English and dropped from the index, so the `fr` hreflang points at a page that will not appear
- C Both pages merge into one URL
- D The canonical is ignored when hreflang is present

10.1.Q8 Transfer MEDIUM What is the purpose of the **`x-default`** hreflang value, and how does it relate to the locale fallback chain?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 10 – International SEO and hreflang

10.1.Q1. `x-default -> https://s.com/` then `distinct from locale URLs: True`

10.1.Q2. `locale variants: 5` then `distinct languages: ['de', 'en', 'fr']`

10.1.Q3. `malformed: ['EN-us', 'e-US', 'en-USA', 'en--US']` then `count: 4`

10.1.Q4. `weights: [(1.0, 'en-US'), (0.9, 'fr-FR'), (0.5, 'de')]` then `chosen:` en-US

10.1.Q5. B language subtag `fr` is lowercase 639-1; region subtag `CA` is uppercase 3166-1 alpha-2.

10.1.Q6. B hreflang must be bidirectional; without B's return tag to A the engine ignores the pairing.

10.1.Q7. B a cross-locale canonical tells Google to drop the French version; keep each locale self-canonical.

10.1.Q8. It names the page to serve users whose language/region matches none of your locales (usually a neutral selector or root). It is the final step of the fallback chain: exact locale → bare language → x-default.

Chapter 11 · Off-Page, Links and Authority

Model internal link equity with a toy PageRank, anchor-text distribution and backlink-profile metrics.

Chapter Skill Map

What this chapter adds: In Off-Page, Links and Authority, the focus is model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics. You practise it across 32 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 11 assumes the reading habits from Chapter 10: International SEO and hreflang. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 12, Local and Map SEO, reuses this chapter's discipline with a new kind of problem.

1 **Projects 371-380**	2 **Projects 381-390**	3 **Projects 391-402**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

A link is a vote, and authority is what flows along the edges of that vote graph.

on-page is what you say; off-page is who vouches for you

On-page SEO is everything you control on your own pages. Off-page SEO is everything other pages say *about* you — and the loudest thing they can say is a link. Treat each link as a vote: a page that links to you is spending a little of its own credibility to say "this is worth seeing." The more credible the voter, and the more independent voters there are, the more authority pools on the target.

The same idea runs *inside* your own site. Every internal link is a vote you cast for one of your own pages, so the link graph quietly decides which of your URLs collects the most equity. The first off-page measurement is therefore pure graph bookkeeping: for each page, how many links point *at* it (in-degree) versus how many it sends *out* (out-degree). The gap tells you where equity gathers.

```
links = {
    'home': ['about', 'blog'],
    'about': ['home'],
    'blog': ['home', 'post1'],
    'post1': ['blog'],
}
out = {p: len(t) for p, t in links.items()}
inc = {p: 0 for p in links}
for targets in links.values():
    for t in targets:
        inc[t] += 1
```

```
for p in sorted(links):
    print(f'{p}: in={inc[p]} out={out[p]}')
```

```
output
about: in=1 out=1
blog: in=2 out=2
home: in=2 out=2
post1: in=1 out=1
```

Read the counts and the homepage already stands out — it collects the most inbound links, so it is the page your own site is implicitly voting most important. That is no accident: a sensible site funnels equity toward its hub.

the page with the highest in-degree is your implicit "most important" URL

Counting edges is only the start. Authority does not just *land* on a page — it flows through it, because a page hands its votes onward to whatever it links to. Following that flow until it settles is exactly what PageRank does, and it is the spine of this chapter. The off-page craft is then: get more independent, high-quality votes pointing in, route your own internal votes toward the pages that matter, and keep the profile looking natural so it reads as earned rather than bought.

IN-DEGREE / OUT-DEGREE

In-degree is the number of links pointing *at* a page; out-degree is the number it sends *out*. High in-degree means equity collects there; high out-degree means it leaks away.

LINK EQUITY ("LINK JUICE")

The ranking value a link passes from source to target. A page splits its equity across its out-links, so each extra out-link dilutes the share every link carries.

PAGERANK

A score for each page equal to the long-run share of a random surfer's visits. Computed by repeatedly spreading rank along out-links (the *power method*) with a damping factor for teleportation, until the values converge.

DAMPING FACTOR (D)

The probability (classically 0.85) that the random surfer follows a link rather than teleporting to a random page. The `(1 − d)` slice is shared out as a baseline so every page keeps some rank and the totals stay conserved.

ANCHOR TEXT

The visible, clickable words of a link. Search engines read it as a description of the target, so a profile of anchors should be a natural *spread* (branded, generic, naked-URL, partial) rather than a pile of one exact-match phrase.

DOFOLLOW VS NOFOLLOW

A `dofollow` link passes equity and is a real vote; a `rel="nofollow"` link is a hint that signals interest but does not vote. Only dofollow links move PageRank.

REFERRING DOMAIN

A distinct site that links to you. A hundred links from one domain count for far less than one link each from a hundred domains, so referring-domain count is the headline backlink metric.

Links are votes; in-degree finds the hub

If a link is a vote, the most-voted-for page is wherever inbound links pile up. Counting in-degree across the internal graph surfaces that hub in one pass — usually the homepage, because everything links back to it. Knowing the hub tells you which URL your own structure is promoting, whether or not that is the page you *want* promoted.

```
from collections import Counter

links = {
    'home': ['about', 'blog', 'shop'],
    'about': ['home'],
    'blog': ['home', 'shop'],
    'shop': ['home'],
    'post1': ['shop'],
}
indeg = Counter()
for targets in links.values():
    for t in targets:
        indeg[t] += 1
top = max(sorted(indeg), key=lambda p: indeg[p])
print(f'most-linked: {top} ({indeg[top]} inbound)')
```

```
output
most-linked: home (3 inbound)
```

Sorting *all* pages by in-degree gives the same idea as a ranked list — the top of it should be your most important pages. If a money page sits near the bottom, your internal linking is voting for the wrong URLs and you fix it by

adding edges, not content.

Equity dilutes: a page splits its vote across out-links

A page does not give each out-link a full vote — it splits its equity *evenly* across them. Two out-links each get half; ten out-links each get a tenth. That is why stuffing a page with links weakens every one of them, and why a `dangling node` (a page with zero out-links) is special: it has nowhere to send its share.

```
rank = {'home': 0.4, 'a': 0.3, 'b': 0.3}
links = {
    'home': ['a', 'b', 'shop'],
    'a': ['home'],
    'b': [],
}
for p, outs in links.items():
    if not outs:
        print(f'{p}: dangling, equity stays put')
        continue
    each = round(rank[p] / len(outs), 3)
    for t in outs:
        print(f'{p} -> {t}: {each}')
```

```
output
home -> a: 0.133
home -> b: 0.133
home -> shop: 0.133
a -> home: 0.3
b: dangling, equity stays put
```

`home` had 0.4 of equity and three out-links, so each link carries 0.133 — add a fourth out-link and that drops again. This even-split rule is the single most important mechanic in the chapter: it is what makes the iteration below converge. *more out-links = thinner equity per link*

One power-method step, then iterate to convergence

PageRank starts every page with equal rank (`1/N`) and repeats one move: each page hands its rank, split evenly, to the pages it links to. A single pass already shows equity starting to pool. Watch `c` — both `a` and `b` point at it, so after one step it holds 0.5:

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c'],
    'c': ['home'],
}
N = len(links)
rank = {p: round(1.0 / N, 3) for p in links}
nxt = {p: 0.0 for p in links}
```

```
for p, outs in links.items():
    share = rank[p] / len(outs)
    for t in outs:
        nxt[t] += share
for p in sorted(nxt):
    print(f'{p}: {round(nxt[p], 3)}')
```

```
output
a: 0.125
b: 0.125
c: 0.5
home: 0.25
```

One step is not the answer — the values are still moving. Real PageRank repeats the step with a damping factor `d = 0.85`: with 85% probability the surfer follows a link, and with 15% they teleport, so each round seeds every page with a baseline `(1 - d)/N` before the link flow is added. On a tiny graph a fixed 50 iterations is far more than enough to settle, and because the process is deterministic the numbers are reproducible every run:

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c'],
    'c': ['home'],
}
N = len(links)
d = 0.85
rank = {p: 1.0 / N for p in links}
for _ in range(50):
    nxt = {p: (1 - d) / N for p in links}
    for p, outs in links.items():
        share = rank[p] / len(outs)
        for t in outs:
            nxt[t] += d * share
    rank = nxt
for p in sorted(rank):
    print(f'{p}: {round(rank[p], 4)}')
print('sum:', round(sum(rank.values()), 4))
```

```
output
a: 0.1736
b: 0.1736
c: 0.3326
home: 0.3203
sum: 1.0
```

Two things to notice. First, the ranks sum to 1.0 — equity is conserved, it only moves around. Second, `c` and `home` end up strongest because the link flow loops through them, while `a` and `b` are mere pass-throughs. There is no

randomness in the loop: same graph, same iterations, same answer.

fixed d, fixed iterations → deterministic, reproducible ranks

A dangling node would break conservation — its rank vanishes each round since it has no out-links. The standard fix collects the dangling rank and redistributes it evenly across all pages, so the total stays at 1.0:

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': [],
    'c': ['home'],
}
N = len(links)
d = 0.85
rank = {p: 1.0 / N for p in links}
for _ in range(50):
    leaked = sum(rank[p] for p, o in links.items() if not o)
    nxt = {p: (1 - d) / N + d * leaked / N for p in links}
    for p, outs in links.items():
        if outs:
            share = rank[p] / len(outs)
            for t in outs:
                nxt[t] += d * share
    rank = nxt
for p in sorted(rank):
    print(f'{p}: {round(rank[p], 4)}')
print('sum:', round(sum(rank.values()), 4))
```

```
output
a: 0.2138
b: 0.2138
c: 0.2646
home: 0.3079
sum: 1.0
```

Anchor text: a healthy profile is a spread, not a pile

Search engines read a link's anchor text as a description of the target. Natural backlinks come with varied anchors — your brand, the bare URL, generic "click here", and the occasional keyword. So you bucket anchors into types and look at the *distribution*: exact-match, partial, branded, generic, naked-url.

```
from collections import Counter

anchors = [
    'buy running shoes', 'click here', 'example.com',
    'running shoes', 'Acme Store', 'best running shoes',
    'read more', 'Acme', 'shoes', 'https://example.com/shop',
]
```

```
target_kw = 'running shoes'
brand = {'acme', 'acme store'}
def classify(a):
    low = a.lower()
    if low == target_kw:
        return 'exact'
    if low in brand:
        return 'branded'
    if low.startswith('https://') or '.com' in low:
        return 'naked-url'
    if low in {'click here', 'read more'}:
        return 'generic'
    if target_kw in low or any(w in low for w in target_kw.split()):
        return 'partial'
    return 'generic'
tally = Counter(classify(a) for a in anchors)
for kind in sorted(tally):
    print(f'{kind}: {tally[kind]}')
```

```
output
branded: 2
exact: 1
generic: 2
naked-url: 2
partial: 3
```

That spread looks earned. The danger is over-optimization: when too many backlinks use the exact-match keyword, the profile looks bought and risks a penalty. A simple share threshold flags it — here 70% exact-match is far past a 30% line:

```
from collections import Counter

anchors = ['running shoes'] * 7 + ['Acme', 'click here', 'shop now']
target_kw = 'running shoes'
total = len(anchors)
exact = Counter(a.lower() for a in anchors)[target_kw]
pct = round(100 * exact / total, 1)
limit = 30.0
verdict = 'OVER-OPTIMIZED' if pct > limit else 'natural'
print(f'exact-match {exact}/{total} = {pct}% (limit {limit}%) -> {verdict}')
```

```
output
exact-match 7/10 = 70.0% (limit 30.0%) -> OVER-OPTIMIZED
```

Counting votes: dofollow, and domains over links

Not every link is a vote. A `rel="nofollow"` link signals interest but does not pass equity, so the `dofollow-to-nofollow ratio` tells you how much of a profile actually counts:

```
from collections import Counter

backlinks = [
    {'from': 'https://a.com', 'rel': 'dofollow'},
    {'from': 'https://b.com', 'rel': 'nofollow'},
    {'from': 'https://c.com', 'rel': 'dofollow'},
    {'from': 'https://d.com', 'rel': 'dofollow'},
    {'from': 'https://e.com', 'rel': 'nofollow'},
]
c = Counter(b['rel'] for b in backlinks)
df, nf = c['dofollow'], c['nofollow']
ratio = round(df / nf, 2)
print(f'dofollow={df} nofollow={nf} ratio={ratio}:1')
```

```
output
dofollow=3 nofollow=2 ratio=1.5:1
```

The single most important headline metric, though, is `referring domains`, not raw link count. Twenty links from one blog are one vote repeated; one link each from twenty sites is twenty independent votes. Collapse the link list to its distinct hosts and count those:

```
backlinks = [
    {'from': 'https://blogA.com/post1', 'to': '/shoes'},
    {'from': 'https://blogA.com/post2', 'to': '/shoes'},
    {'from': 'https://news.org/review', 'to': '/shoes'},
    {'from': 'https://forum.net/t/9', 'to': '/home'},
    {'from': 'https://blogA.com/post3', 'to': '/home'},
]
def domain(url):
    return url.split('://', 1)[1].split('/', 1)[0]
domains = {domain(b['from']) for b in backlinks}
print('total backlinks:', len(backlinks))
print('referring domains:', len(domains))
for d in sorted(domains):
    print(' ', d)
```

```
output
total backlinks: 5
referring domains: 3
  blogA.com
  forum.net
  news.org
```

Five links collapse to three voters. A profile-health verdict combines these signals: too high an exact-match anchor share *and* too few distinct anchors together mean an unnatural profile — the reverse means an earned one.

domains, dofollow share, anchor spread → one health read

Example 1

A four-page site has links home→[a, b], a→[c], b→[c], c→[home]. Using the even-split rule, run ONE power-method step (no damping) from equal starting rank 0.25 each. What rank does page c hold after the step?

SOLUTION

1. Start equal: every page has rank `1/4 = 0.25`.
2. Each page splits its rank evenly over its out-links. a has one out-link (→c), so it sends all 0.25 to c; b likewise sends its 0.25 to c.
3. Page c receives 0.25 from a + 0.25 from b = 0.5. Meanwhile home gets c's 0.25 (rounded), and a, b each get half of home's 0.25 = 0.125.

c holds 0.5 after one step — two pages funnel their whole vote into it. Equity is already pooling on the well-linked page.

one step shows the trend; iterate to get the final value

Example 2

A backlink profile has these anchors: "running shoes" ×4, "Acme", "click here", "shop now", "best shoes" (8 total). The target keyword is "running shoes". Is this profile healthy on the two anchor signals (exact-match ≤30%, unique-ratio ≥0.5)?

SOLUTION

1. Exact-match share: "running shoes" appears 4 times in 8 anchors → 4/8 = 50%, which is over the 30% limit.
2. Anchor diversity: the distinct anchors are {running shoes, acme, click here, shop now, best shoes} = 5 unique out of 8 → 5/8 = 0.625, which clears the 0.5 floor.
3. One signal fails (exact-match over-optimised) and one passes, so any failing flag makes the verdict RISKY.

RISKY — exact-match over-optimised at 50%. Diversity is fine, but one bad signal is enough; the fix is to earn more branded/generic/naked-url anchors.

a single tripped flag overrides a passing one

Example 3

Site X has backlinks from blogA.com (3 links), blogB.com (2 links), and news.org (1 link) — 6 links total. Competitor Y also has a link from forbes.com. By the headline metric, how many independent votes does X have, and what is the one actionable gap?

SOLUTION

1. Collapse X's 6 links to distinct hosts: {blogA.com, blogB.com, news.org} → 3 referring domains, not 6.
2. Referring domains, not raw links, is the headline metric, so X has 3 independent votes.
3. Set-subtract X's domains from Y's: forbes.com links to Y but not X, so it is the link gap.

3 referring domains; the actionable gap is forbes.com (links to the competitor but not to you).

domains over links; gap = competitor's domains minus yours

Mental-Math Shortcut

Count the voters, not the votes — then watch where the votes flow. One link each from ten sites beats ten links from one site, because authority rewards *independent* endorsement. And inside your own site, every internal link is a vote you control: route them toward the pages that matter (high in-degree), keep out-links lean so each carries more equity, and let PageRank's even-split flow do the rest. Natural beats optimised — a varied anchor profile from many domains always outranks a stack of exact-match links bought in bulk.

⚠ Common Mistakes

- Bragging about raw backlink count when the referring-domain count is tiny — 500 links from 3 sites is 3 votes, not 500.
- Chasing exact-match keyword anchors. Push the exact-match share past ~30% and the profile reads as manipulated, inviting a penalty instead of a boost.
- Treating nofollow links as ranking votes. They can drive traffic and signal interest, but they pass no equity, so they do not move PageRank.
- Adding internal links indiscriminately. Because a page splits its equity evenly across out-links, every extra out-link thins the share each one carries — link with intent, not abundance.
- Leaving important pages orphaned (zero in-degree). With no internal link pointing at them they receive almost no equity and may never be indexed.

This chapter contains 32 hands-on mini-projects (Projects 371–402). Type each one out, run it, and experiment.

Project 371 Tally In-Degree and Out-Degree per Page

The Idea Behind It

A link graph is a dictionary mapping each page to the list of pages it points at. This project measures two opposite quantities: out-degree is just the length of each target list, while in-degree is built by walking every

target list and incrementing a counter for the page it lands on. The gap between the two numbers tells you which pages collect equity versus which spend it.

What You'll Build & Learn

Compute inbound and outbound link counts for every page in a small internal link graph and print them side by side.

- In-degree versus out-degree on a link graph
- Counting inbound links by reversing the edges
- Dictionary comprehensions for per-page tallies

🧭 Skill Roadmap

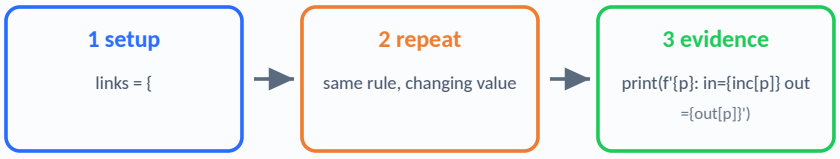
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 372 changes the surface problem while preserving the underlying method. *progress check*

Reading target: tally in-degree and out-degree per page is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{p}: in={inc[p]} out={out[p]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`out = {p: len(t) for p, t in links.items()}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{p}: in={inc[p]} out={out[p]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `out = {p: len(t) for p, t in links.items()}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Tally In-Degree and Out-Degree per Page: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links, home, about, blog, post1. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: links = { | Core: out = {p: len(t) for p, t in links.items()} | Evidence: print(f'{p}: in={inc[p]} out={out[p]}')

Reader to Writer Bridge

1. Read: explain tally in-degree and out-degree per page in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 372 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{p}: in={inc[p]} out={out[p]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of tally in-degree and out-degree per page and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

From Concept to Code (Python)

Out-degree is just the length of each adjacency list, while in-degree needs a second pass that counts every page as it appears as a link target.

```
links = {
    'home': ['about', 'blog'],
    'about': ['home'],
    'blog': ['home', 'post1'],
    'post1': ['blog'],
}
out = {p: len(t) for p, t in links.items()}
inc = {p: 0 for p in links}
for targets in links.values():
    for t in targets:
        inc[t] += 1
for p in sorted(links):
    print(f'{p}: in={inc[p]} out={out[p]}')
```

```
output
about: in=1 out=1
blog: in=2 out=2
```

```
home: in=2 out=2
post1: in=1 out=1
```

How it works:

1. The links dict gives each page a list of outbound targets, like home pointing at about and blog.
2. A comprehension sets out[p] to len(t), the number of outbound links each page emits.
3. inc starts every page at zero, then nested loops walk each target list and bump inc[t] by one.
4. Iterating over sorted(links) prints each page with its in and out counts aligned.

Try it:

1. Add a third metric that computes the ratio of in-degree to out-degree per page
2. Extend the links dict with a deeper blog structure and rerun the tallies
3. Print only pages whose in-degree exceeds their out-degree by two or more
4. Add a link from `post1` to `home` and watch `home`'s in-degree climb.

⚠ Common Mistakes

- Reading `len(links[p])` as in-degree actually gives out-degree.
- Initialising `inc` only from targets drops pages that have zero inbound links.
- Iterating `links.keys()` instead of the value lists never sees the link targets.

Skills gained link graphs | in-degree | dict comprehension

→ Where You'll Use This: Internal link audits in tools like Screaming Frog report exactly these numbers, helping you spot pages that hoard equity or leak it without earning links back.

⚡ Mental Model

Think of each page as a town: out-degree is roads leaving it, in-degree is roads arriving. Towns with many arriving roads become hubs.

Project 372 Find the Site's Internal Link Hub

💡 The Idea Behind It

Building on raw in-degree counts, this project finds the single page that the rest of the site links to most. It accumulates inbound counts with a Counter, then uses max over a sorted key list so ties resolve deterministically by name. The winner is the URL your own internal linking is implicitly nominating as most important, almost always the homepage.

What You'll Build & Learn

Identify the most internally linked page so you know which URL your site treats as its authority hub.

- Using Counter to accumulate inbound link totals
- max with a key function over sorted keys
- Deterministic tie-breaking by sorting first

Skill Roadmap

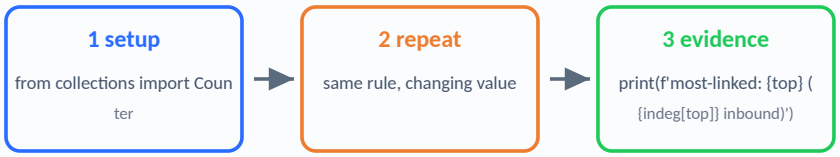
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 371: tally in-degree and out-degree per page. Project 373 changes the surface problem while preserving the underlying method. *progress check*

Reading target: find the site's internal link hub is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'most-linked: {top} ({indeg[top]} inbound)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.

Move	Code to inspect	Why it matters
2. Apply the rule	`for targets in links.values():`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'most-linked: {top} ({indeg[top]} inbound)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for targets in links.values():`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find the Site's Internal Link Hub: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, links, home, about. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: for targets in links.values(): | Evidence: print(f'most-linked: {top} ({indeg[top]} inbound)')

Reader to Writer Bridge

1. Read: explain find the site's internal link hub in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 371.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 373 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'most-linked: {top} ({indeg[top]} inbound)')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find the site's internal link hub and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

After tallying in-degree, `max` over the sorted keys with an in-degree key returns the hub, with sorting making any tie break alphabetically and deterministically.

```
from collections import Counter

links = {
    'home': ['about', 'blog', 'shop'],
    'about': ['home'],
    'blog': ['home', 'shop'],
    'shop': ['home'],
    'post1': ['shop'],
}
indeg = Counter()
for targets in links.values():
    for t in targets:
        indeg[t] += 1
top = max(sorted(indeg), key=lambda p: indeg[p])
print(f'most-linked: {top} ({indeg[top]} inbound)')
```

```
output
most-linked: home (3 inbound)
```

How it works:

1. A Counter named indeg starts empty and is filled by looping over every target list in the links dict.
2. Each appearance of a page as a link target increments its inbound count by one.

3. max scans sorted(indeg) using a lambda that returns indeg[p], picking the page with the highest count.
4. The print reports the winning page and its inbound link total, here home with three.

Try it:

1. Return the top three most-linked pages instead of only the single winner
2. Break ties by preferring the page with the shortest name
3. Flag a warning when the most-linked page is not named home
4. Add two more links to `shop` to flip the winner away from `home`.

⚠ Common Mistakes

- Calling `max` over the unsorted Counter makes ties resolve unpredictably.
- Passing `indeg.values()` to `max` loses which page the count belongs to.
- A page that is never a link target is absent from the Counter, not zero.

Skills gained Counter argmax internal linking

→ Where You'll Use This: When auditing a site, confirming the most-linked page matches your intended priority page catches accidental cases where a thin page outranks your money page internally.

⚡ Mental Model

It is like counting how many roads in a town lead to one building. The busiest intersection is what locals treat as the center.

Project 373 Spread Rank Once with the Power Method

💡 The Idea Behind It

This introduces the core PageRank move without the bells and whistles. Every page begins with equal rank, one over the page count. A single pass then has each page divide its rank evenly among its outbound links and hand each share to the target. After one iteration you can already see equity starting to pool on the well-linked page c, which two pages feed.

What You'll Build & Learn

Run one power-method step of PageRank so you can watch equity begin to redistribute across a link graph.

- Initialising equal rank as one over N
- Dividing a page's rank across its out-links
- One iteration of the power method

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 372: find the site's internal link hub. Project 374 changes the surface problem while preserving the underlying method.

progress check

Reading target: spread rank once with the power method is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{p}: {round(nxt[p], 3)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`rank = {p: round(1.0 / N, 3) for p in links}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{p}: {round(nxt[p], 3)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rank = {p: round(1.0 / N, 3) for p in links}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Spread Rank Once with the Power Method: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links, home, len, rank, round. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: links = { | Core: rank = {p: round(1.0 / N, 3) for p in links} | Evidence: print(f'{p}: {round(nxt[p], 3)}')

Reader to Writer Bridge

1. Read: explain spread rank once with the power method in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 372.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 374 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{p}: {round(nxt[p], 3)}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of spread rank once with the power method and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each page divides its current rank by its out-degree and adds that share to every target; one iteration is the building block the full algorithm repeats.

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c'],
    'c': ['home'],
}
N = len(links)
rank = {p: round(1.0 / N, 3) for p in links}
nxt = {p: 0.0 for p in links}
for p, outs in links.items():
    share = rank[p] / len(outs)
    for t in outs:
        nxt[t] += share
for p in sorted(nxt):
    print(f'{p}: {round(nxt[p], 3)}')
```

```
output
a: 0.125
b: 0.125
c: 0.5
home: 0.25
```

How it works:

1. rank gives each of the four pages a starting value of 0.25, rounded for display.
2. nxt is a fresh dict of zeros that will receive the redistributed equity.
3. For each page, share is its rank divided by the number of out-links, added to every target in nxt.
4. Printing sorted(nxt) shows c at 0.5 because both a and b funnel their share into it.

Try it:

1. Run the redistribution step three times in a loop and print after each pass

2. Add a new page that links only to home and observe the change
3. Print the total of all ranks after the pass to confirm it stays near one
4. Run a second iteration by copying `nxt` back into `rank` and repeating.

⚠ Common Mistakes

- Updating `rank` in place instead of writing to a fresh `nxt` mixes old and new values.
- Dividing by `len(links)` rather than the page's own out-degree spreads rank wrongly.
- A page with zero out-links would divide by zero, so this fixture gives every page an out-link.

Skills gained PageRank | power method | rank distribution

→ Where You'll Use This: Understanding this single step demystifies why link-equity flow tools show certain pages ballooning, and it underpins every internal-linking recommendation an SEO makes.

Mental Model

Each page pours a full cup of water evenly into the cups it points at. After one pour, cups fed by many start filling up.

Project 374 Iterate PageRank to Convergence with Damping

The Idea Behind It

The full algorithm adds two refinements to the single pass. A damping factor of 0.85 models the chance a surfer keeps clicking, while the remaining fifteen percent teleports uniformly to every page as the baseline term. Repeating the step fifty times lets the values settle, and summing them confirms the ranks still total one, the signature of a correct PageRank implementation.

What You'll Build & Learn

Iterate PageRank with damping until the ranks converge, and verify the total equity stays conserved at one.

- The 0.85 damping factor and teleport term
- Iterating the power method to convergence
- Checking ranks sum to one

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 373: spread rank once with the power method. Project 375 changes the surface problem while preserving the underlying method. *progress check*

Reading target: iterate pagerank to convergence with damping is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print('sum:', round(sum(rank.values()), 4))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`rank = {p: 1.0 / N for p in links}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('sum:', round(sum(rank.values()), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rank = {p: 1.0 / N for p in links}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Iterate PageRank to Convergence with Damping: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links, home, len, rank, range. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: links = { | Core: rank = {p: 1.0 / N for p in links} | Evidence: print('sum:', round(sum(rank.values()), 4))

🛠️ Reader to Writer Bridge

1. Read: explain iterate pagerank to convergence with damping in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 373.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 375 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('sum:', round(sum(rank.values()), 4))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of iterate pagerank to convergence with damping and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Each round gives every page a `(1-d)/N` teleport baseline then adds `d` times the rank flowing in along links; repeating drives the vector to its stationary distribution that sums to 1.

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c'],
    'c': ['home'],
}
N = len(links)
d = 0.85
rank = {p: 1.0 / N for p in links}
for _ in range(50):
    nxt = {p: (1 - d) / N for p in links}
    for p, outs in links.items():
        share = rank[p] / len(outs)
        for t in outs:
            nxt[t] += d * share
    rank = nxt
for p in sorted(rank):
    print(f'{p}: {round(rank[p], 4)}')
print('sum:', round(sum(rank.values()), 4))
```

```
output
a: 0.1736
b: 0.1736
c: 0.3326
home: 0.3203
sum: 1.0
```

How it works:

1. d is set to 0.85 and rank starts uniform at one over N for the four pages.
2. Each of fifty iterations seeds nxt with the teleport baseline (1-d)/N for every page.
3. Inner loops add d times each page's per-link share to its targets, then rank becomes nxt.
4. After the loop, sorted ranks print and sum(rank.values()) confirms a total of 1.0.

Try it:

1. Lower the damping factor to 0.5 and compare how the converged ranks shift

2. Stop the loop early once ranks change by less than a tiny threshold
3. Add a fifth page and watch how it dilutes everyone else's share
4. Drop the damping factor to 0.5 and watch the ranks flatten toward uniform.

⚠ Common Mistakes

- Forgetting the `(1-d)/N` teleport term lets rank drain into sinks and stop summing to 1.
- Mutating `rank` mid-round uses half-updated values and breaks convergence.
- Too few iterations on a larger graph stops before the ranks settle.

Skills gained damping factor | convergence | PageRank

→ Where You'll Use This: This is the actual model search engines descend from, so running it on your own crawl reveals which internal pages would accumulate the most algorithmic authority.

Mental Model

Picture a wanderer clicking links 85 percent of the time and randomly jumping the rest. Where they spend most time is a page's PageRank.

Project 375 Split Link Equity Evenly Across Out-Links

The Idea Behind It

Here the focus narrows to one rule of equity flow: a page splits its rank into equal slices, one per outbound link, so more links mean thinner slices. The code prints the per-link share and handles the special case of a dangling page with no out-links, which keeps its equity rather than passing anything on. It makes the dilution cost of stuffing a page with links visible.

What You'll Build & Learn

Show the exact equity slice each outbound link receives, and demonstrate what happens at a dangling page.

- Equity splits equally across out-links
- More links means a thinner share each
- Recognising a dangling page with no out-links

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 374: iterate pagerank to convergence with damping. Project 376 changes the surface problem while preserving the underlying method. *progress check*

Reading target: split link equity evenly across out-links is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `rank = {'home': 0.4, 'a': 0.3, 'b': 0.3}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{p} -> {t}: {each}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`rank = {'home': 0.4, 'a': 0.3, 'b': 0.3}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`if not outs:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{p} -> {t}: {each}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not outs:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Split Link Equity Evenly Across Out-Links: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: rank, home, links, shop, outs. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: rank = {'home': 0.4, 'a': 0.3, 'b': 0.3} | Core: if not outs: | Evidence: print(f'{p} -> {t}: {each}')

🛠️ Reader to Writer Bridge

1. Read: explain split link equity evenly across out-links in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 374.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 376 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{p} -> {t}: {each}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `rank = {'home': 0.4, 'a': 0.3, 'b': 0.3}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split link equity evenly across out-links and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Dividing a page's equity by its out-degree gives each link an equal share, while a page with no out-links is a dangling node whose equity has nowhere to go.

```
rank = {'home': 0.4, 'a': 0.3, 'b': 0.3}
links = {
    'home': ['a', 'b', 'shop'],
    'a': ['home'],
    'b': [],
}
for p, outs in links.items():
    if not outs:
        print(f'{p}: dangling, equity stays put')
        continue
    each = round(rank[p] / len(outs), 3)
    for t in outs:
        print(f'{p} -> {t}: {each}')
```

```
output
home -> a: 0.133
home -> b: 0.133
home -> shop: 0.133
a -> home: 0.3
b: dangling, equity stays put
```

How it works:

1. rank holds preset equity values for home, a, and b, while links lists each page's targets.
2. The loop checks if outs is empty and, if so, prints that the page is dangling and skips it.
3. Otherwise each is computed as rank[p] divided by the out-link count, rounded to three places.
4. home with three links gives each target 0.133, while a with one link passes its full 0.3.

Try it:

1. Add a fourth target to home and watch every share shrink accordingly
2. Give the dangling page b a single out-link and rerun
3. Sort the printed shares from largest to smallest before display
4. Give `b` an out-link and watch its equity start to flow instead of stalling.

⚠ Common Mistakes

- Dividing by the total page count instead of the out-degree under-shares the equity.
- Skipping the empty-list guard divides by zero on the dangling page `b`.
- Rounding before summing the shares can make them not quite add back to the source rank.

Skills gained equity sculpting | link dilution | dangling node

→ Where You'll Use This: When deciding how many links to place on a page, this shows why a cluttered navigation dilutes the equity reaching each linked page, a real consideration in template design.

⚡ Mental Model

A page slices a pizza equally for everyone it links to. Invite more guests and each slice gets smaller. A dangling page eats alone.

Project 376 Handle a Dangling Node in PageRank

💡 The Idea Behind It

A dangling node breaks naive PageRank because its rank would evaporate each round. This version fixes the leak: before redistributing, it sums the rank held by all dead-end pages and sprinkles that pooled equity uniformly across every page through the d*leaked/N term. The sum check printing 1.0 proves the conservation that the leak would otherwise destroy.

What You'll Build & Learn

Patch PageRank so a dangling node's equity is redistributed instead of lost, keeping the total conserved.

- Why dangling nodes leak PageRank
- Summing and redistributing dangling rank
- Equity conservation across iterations

🧭 Skill Roadmap

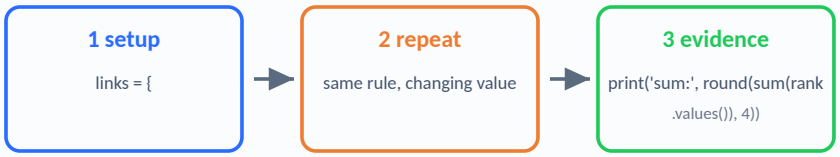
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 375: split link equity evenly across out-links. Project 377 changes the surface problem while preserving the underlying method. *progress check*

Reading target: handle a dangling node in pagerank is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print('sum:', round(sum(rank.values()), 4))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`leaked = sum(rank[p] for p, o in links.items() if not o)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('sum:', round(sum(rank.values()), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `leaked = sum(rank[p] for p, o in links.items() if not o)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Handle a Dangling Node in PageRank: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links, home, len, rank, range. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: links = { | Core: leaked = sum(rank[p] for p, o in links.items() if not o) | Evidence: print('sum:', round(sum(rank.values()), 4))

🛠️ Reader to Writer Bridge

1. Read: explain handle a dangling node in pagerank in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 375.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 377 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('sum:', round(sum(rank.values()), 4))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of handle a dangling node in pagerank and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Summing the rank sitting on dangling pages and sprinkling `d * leaked / N` onto every page each round patches the leak so the rank vector keeps summing to 1.

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': [],
    'c': ['home'],
}
N = len(links)
d = 0.85
rank = {p: 1.0 / N for p in links}
for _ in range(50):
    leaked = sum(rank[p] for p, o in links.items() if not o)
    nxt = {p: (1 - d) / N + d * leaked / N for p in links}
    for p, outs in links.items():
        if outs:
            share = rank[p] / len(outs)
            for t in outs:
                nxt[t] += d * share
    rank = nxt
for p in sorted(rank):
    print(f'{p}: {round(rank[p], 4)}')
print('sum:', round(sum(rank.values()), 4))
```

```
output
a: 0.2138
b: 0.2138
c: 0.2646
home: 0.3079
sum: 1.0
```

How it works:

1. Page b has an empty out-list, marking it as the dangling node in this graph.
2. Each iteration computes leaked, the sum of rank over pages with no out-links.
3. nxt seeds every page with the teleport term plus d times leaked spread over N.
4. Pages with out-links still add their normal share, and the final sum prints exactly 1.0.

Try it:

1. Add a second dangling page and confirm the sum still equals one
2. Print the leaked value each iteration to watch it stabilise
3. Give b one out-link and verify the leaked term drops to zero
4. Remove the leak redistribution and watch the rank sum drift below 1.

⚠ Common Mistakes

- Ignoring the leak lets total rank fall below 1 every iteration.
- Computing `leaked` from the new ranks instead of the previous round's double-counts.
- Treating the dangling page as if it linked to itself changes the result subtly.

Skills gained dangling node | rank conservation | PageRank

→ Where You'll Use This: Real sites are full of dead-end pages like terminal PDFs or contact forms, so handling them correctly is essential before trusting any internal PageRank estimate.

Mental Model

A dead-end page is a bucket with no spout. Instead of letting its water sit useless, you ladle it back evenly into every bucket on the site.

Project 377 Tally the Anchor-Text Distribution

The Idea Behind It

Backlink anchor text is not one thing but a mix of categories, and this classifier sorts a list of anchors into exact-match, partial, branded, generic, and naked-url buckets. The classify function applies ordered rules, lowercasing first, and a Counter tallies the distribution. The spread it produces is the signal: a natural profile shows variety rather than a tower of exact-match phrases.

What You'll Build & Learn

Classify a list of backlink anchors into type buckets and tally how the profile is distributed.

- Anchor types: exact, partial, branded, generic, naked-url
- Ordered if-rules for classification
- Tallying categories with Counter

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 376: handle a dangling node in pagerank. Project 378 changes the surface problem while preserving the underlying method.

progress check

Reading target: tally the anchor-text distribution is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{kind}: {tally[kind]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`if low == target_kw:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{kind}: {tally[kind]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if low == target_kw:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Tally the Anchor-Text Distribution: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, anchors, buy, running. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: if low == target_kw: | Evidence: print(f'{kind}: {tally[kind]}')

🛠️ Reader to Writer Bridge

1. Read: explain tally the anchor-text distribution in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 376.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 378 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{kind}: {tally[kind]}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of tally the anchor-text distribution and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A small classifier maps each anchor to one bucket by checking it against the target keyword, brand names, generic phrases, and URL markers, then a Counter tallies the spread.

```
from collections import Counter

anchors = [
    'buy running shoes', 'click here', 'example.com',
    'running shoes', 'Acme Store', 'best running shoes',
    'read more', 'Acme', 'shoes', 'https://example.com/shop',
]
target_kw = 'running shoes'
brand = {'acme', 'acme store'}
def classify(a):
    low = a.lower()
    if low == target_kw:
        return 'exact'
    if low in brand:
        return 'branded'
    if low.startswith('https://') or '.com' in low:
        return 'naked-url'
    if low in {'click here', 'read more'}:
        return 'generic'
    if target_kw in low or any(w in low for w in target_kw.split()):
        return 'partial'
    return 'generic'
tally = Counter(classify(a) for a in anchors)
for kind in sorted(tally):
    print(f'{kind}: {tally[kind]}')
```

```
output
branded: 2
exact: 1
generic: 2
naked-url: 2
partial: 3
```

How it works:

1. A list of ten anchors and a target_kw of running shoes feed the classifier.

2. classify lowercases each anchor, then checks exact match, brand set membership, and URL markers in order.
3. Partial matches are caught when the target phrase or any of its words appears in the anchor.
4. Counter tallies the returned labels, printing each kind sorted with its count.

Try it:

1. Add an image-link category for anchors that are empty strings
2. Make the brand set case-insensitive by normalising it once up front
3. Print each category as a percentage of total anchors
4. Add the anchor `running` and confirm it lands in `partial`, not `exact`.

⚠ Common Mistakes

- Checking `partial` before `exact` would mislabel the exact-match anchor as partial.
- Forgetting to lowercase makes `Acme` miss the branded set keyed in lower case.
- Listing the naked-URL test after the partial test catches `.com` anchors as partial.

Skills gained anchor text | classification | Counter

→ Where You'll Use This: Backlink tools report anchor distributions exactly like this, and recognising an unhealthy skew toward exact-match anchors is a routine part of a backlink audit.

Mental Model

Sorting anchors is like sorting mail into pigeonholes by type. A healthy mailbox has a balanced spread, not a hundred identical flyers.

Project 378 Detect an Over-Optimized Anchor Profile

The Idea Behind It

This zooms in on a single danger signal from the anchor mix: an oversized share of exact-match anchors. It counts how many backlinks use the target keyword verbatim, turns that into a percentage of the total, and compares it against a thirty-percent threshold to label the profile over-optimised or natural. The arithmetic is trivial but the judgment it encodes prevents penalties.

What You'll Build & Learn

Flag a backlink profile as over-optimised when its exact-match anchor share crosses a safe threshold.

- Exact-match anchor share as a risk signal
- Turning a count into a percentage

- Threshold comparison for a verdict

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 377: tally the anchor-text distribution. Project 379 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect an over-optimized anchor profile is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'exact-match {exact}/{total} = {pct}% (limit {limit}%) -> {verdict}')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`verdict = 'OVER-OPTIMIZED' if pct > limit else 'natural'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'exact-match {exact}/{total} = {pct}% (limit {limit}%) -> {verdict}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `verdict = 'OVER-OPTIMIZED' if pct > limit else 'natural'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect an Over-Optimized Anchor Profile: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, anchors, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import Counter | Core: verdict = 'OVER-OPTIMIZED' if pct > limit else 'natural' | Evidence: print(f'exact-match {exact}/{total} = {pct}% (limit {limit}%) -> {verdi...

Reader to Writer Bridge

1. Read: explain detect an over-optimized anchor profile in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 377.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 379 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'exact-match {exact}/{total} = {pct}% (limit {limit}%) -> {verdict}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect an over-optimized anchor profile and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Counting how many anchors equal the target keyword and dividing by the total gives the exact-match share, which over a 30% threshold trips the over-optimized flag.

```
from collections import Counter

anchors = ['running shoes'] * 7 + ['Acme', 'click here', 'shop now']
target_kw = 'running shoes'
total = len(anchors)
exact = Counter(a.lower() for a in anchors)[target_kw]
pct = round(100 * exact / total, 1)
limit = 30.0
verdict = 'OVER-OPTIMIZED' if pct > limit else 'natural'
print(f'exact-match {exact}/{total} = {pct}% (limit {limit}%) -> {verdict}')
```

```
output
exact-match 7/10 = 70.0% (limit 30.0%) -> OVER-OPTIMIZED
```

How it works:

1. The anchors list is built as seven exact-match phrases plus three varied ones, total ten.
2. A Counter on lowercased anchors gives exact, the count of the target keyword.
3. pct is 100 times exact over total, rounded, and limit is fixed at 30.0.
4. A conditional sets verdict to OVER-OPTIMIZED because 70 percent exceeds the limit, then prints it.

Try it:

1. Lower the limit to 20 percent and see whether the verdict changes
2. Report which specific anchor is over-represented, not just the percentage
3. Add a second branded-anchor floor that warns when branded share is too low
4. Add three branded anchors and watch the percentage fall below the limit.

⚠ Common Mistakes

- Comparing the raw exact count to the limit instead of the percentage ignores profile size.
- Counting anchors that merely contain the keyword inflates the exact-match share.
- Using `>=` rather than `>` flags a profile sitting exactly on the limit.

Skills gained over-optimisation | threshold | anchor risk

→ Where You'll Use This: Before launching a link-building campaign you set anchor-text guardrails, and a check like this stops a writer or vendor from overusing the money keyword and triggering a filter.

⚡ Mental Model

It is a smoke alarm for anchors: when too many links shout the same keyword, the alarm trips before the algorithm decides the smoke is arson.

Project 379 Count Referring Domains From a Backlink List

💡 The Idea Behind It

Raw backlink counts mislead because one site can link a thousand times. This project parses the host out of each backlink URL with simple string splitting and collapses the list into a set of unique referring domains. Comparing the five raw links against three distinct domains makes the headline backlink metric concrete: who links to you matters more than how often.

What You'll Build & Learn

Count distinct referring domains from a backlink list and contrast that with the raw link total.

- Referring domains versus raw link count
- Extracting a host with string splitting
- Deduplicating with a set

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 378: detect an over-optimized anchor profile. Project 380 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count referring domains from a backlink list is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(' ', d)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`domains = {domain(b['from']) for b in backlinks}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' ', d)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `domains = {domain(b['from']) for b in backlinks}`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Count Referring Domains From a Backlink List: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, https, blogA, com, post1. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: backlinks = [| Core: domains = {domain(b['from']) for b in backlinks} | Evidence: print(' ', d)

🛠️ Reader to Writer Bridge

1. Read: explain count referring domains from a backlink list in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 378.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 380 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' ', d)`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count referring domains from a backlink list and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Slicing the host out of each linking URL and collecting them in a set collapses many links from one site into a single referring domain.

```
backlinks = [
    {'from': 'https://blogA.com/post1', 'to': '/shoes'},
    {'from': 'https://blogA.com/post2', 'to': '/shoes'},
    {'from': 'https://news.org/review', 'to': '/shoes'},
    {'from': 'https://forum.net/t/9', 'to': '/home'},
    {'from': 'https://blogA.com/post3', 'to': '/home'},
]
def domain(url):
    return url.split('://', 1)[1].split('/', 1)[0]
domains = {domain(b['from']) for b in backlinks}
print('total backlinks:', len(backlinks))
print('referring domains:', len(domains))
for d in sorted(domains):
    print(' ', d)
```

```
output
total backlinks: 5
referring domains: 3
  blogA.com
  forum.net
  news.org
```

How it works:

1. backlinks is a list of dicts each holding a from URL and a to path.
2. domain splits a URL after the scheme, then before the first slash, returning the host.
3. A set comprehension over every from URL yields the unique referring domains.
4. It prints the total link count, the domain count, and each sorted domain.

Try it:

1. Also count how many backlinks each domain contributed
2. Handle URLs that lack a scheme without raising an error
3. Report the average number of links per referring domain
4. Add a fourth `blogA.com` link and confirm the domain count does not move.

⚠ Common Mistakes

- Counting `len(backlinks)` reports links, not the distinct domains that matter.
- Splitting on `.com` instead of the path breaks on other TLDs and subpaths.
- Forgetting to sort the domain set makes the printed list nondeterministic.

Skills gained referring domains | URL parsing | sets

→ Where You'll Use This: Referring-domain count is the first number on any backlink dashboard, and explaining why it beats raw link count is a constant conversation with clients chasing vanity totals.

Mental Model

A hundred votes from one person is one opinion. A hundred votes from a hundred people is consensus. Domains are the people.

Project 380 Measure the Dofollow-to-Nofollow Ratio

The Idea Behind It

Not every backlink passes ranking power, so this project measures the split between dofollow links that vote and nofollow links that only hint. A Counter tallies the rel value of each backlink, then the dofollow count is divided by the nofollow count to express the balance as a ratio. The single figure summarises how much of a profile is actually working for rankings.

What You'll Build & Learn

Tally dofollow versus nofollow backlinks and express their balance as a single ratio.

- Dofollow links pass equity, nofollow do not
- Tallying a field with Counter
- Expressing a balance as a ratio

Skill Roadmap

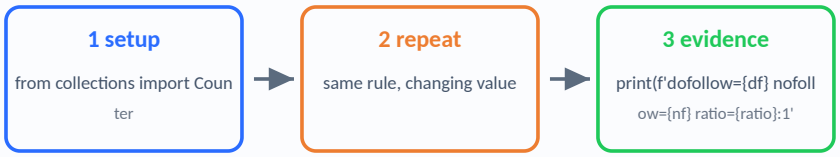
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 379: count referring domains from a backlink list. Project 381 changes the surface problem while preserving the underlying method. *progress check*

Reading target: measure the dofollow-to-nofollow ratio is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'dofollow={df} nofollow={nf} ratio={ratio}:1')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`c = Counter(b['rel'] for b in backlinks)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'dofollow={df} nofollow={nf} ratio={ratio}:1')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `c = Counter(b['rel'] for b in backlinks)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure the Dofollow-to-Nofollow Ratio: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, backlinks, https, com. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: c = Counter(b['rel'] for b in backlinks) | Evidence: print(f'dofollow={df} nofollow={nf} ratio={ratio}:1')

🛠️ Reader to Writer Bridge

1. Read: explain measure the dofollow-to-nofollow ratio in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 379.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 381 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'dofollow={df} nofollow={nf} ratio={ratio}:1')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure the dofollow-to-nofollow ratio and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A Counter over the `rel` field splits the links into dofollow and nofollow, and dividing the two gives the equity-passing ratio.

```
from collections import Counter

backlinks = [
    {'from': 'https://a.com', 'rel': 'dofollow'},
    {'from': 'https://b.com', 'rel': 'nofollow'},
    {'from': 'https://c.com', 'rel': 'dofollow'},
    {'from': 'https://d.com', 'rel': 'dofollow'},
    {'from': 'https://e.com', 'rel': 'nofollow'},
]
c = Counter(b['rel'] for b in backlinks)
df, nf = c['dofollow'], c['nofollow']
ratio = round(df / nf, 2)
print(f'dofollow={df} nofollow={nf} ratio={ratio}:1')
```

```
output
dofollow=3 nofollow=2 ratio=1.5:1
```

How it works:

1. backlinks is a list of dicts each carrying a from URL and a rel attribute.
2. Counter over the rel of every backlink produces counts for dofollow and nofollow.
3. df and nf unpack those two counts from the Counter.
4. ratio is df divided by nf, rounded, and printed as a colon-one figure of 1.5.

Try it:

1. Guard against a zero nofollow count to avoid a division error
2. Add ugc and sponsored as their own rel categories
3. Express the dofollow links as a percentage of all backlinks
4. Add two more dofollow links and watch the ratio climb.

⚠ Common Mistakes

- Dividing nofollow by dofollow inverts the ratio you meant to report.

- A profile with zero nofollow links would divide by zero, so the fixture has both.
- Treating `ugc` and `sponsored` rels as dofollow overstates passed equity.

Skills gained dofollow nofollow ratio Counter

→ **Where You'll Use This:** A profile that is almost entirely nofollow earns trust but little ranking lift, so this ratio frames whether a site needs more equity-passing links in its strategy.

Mental Model

Dofollow links are signed votes that count toward the tally; nofollow links are unsigned notes the judges read but cannot count.

Project 381 Score Linking-Domain Diversity

The Idea Behind It

Diversity here is a ratio rather than a category breakdown. The project extracts the host from each link, collects the unique ones in a set, and divides that count by the total number of links to produce a single diversity score. A result of 0.5 means half the links are essentially repeats from the same handful of sites, a sign the profile lacks breadth.

What You'll Build & Learn

Score how spread out a backlink profile is by dividing unique domains by total links.

- Domain diversity as a ratio
- Set length over list length
- Low diversity signals repeated sources

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 380: measure the dofollow-to-nofollow ratio. Project 382 changes the surface problem while preserving the underlying method.

progress check

Reading target: score linking-domain diversity is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

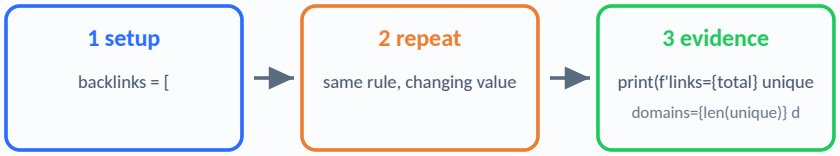
Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'links={total} unique domains={len(unique)} diversity={diversity}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`unique = {domain(u) for u in backlinks}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'links={total} unique domains={len(unique)} diversity={diversity}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `unique = {domain(u) for u in backlinks}`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Score Linking-Domain Diversity: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, https, com, domain, split. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: backlinks = [| Core: unique = {domain(u) for u in backlinks} | Evidence: print(f'links={total} unique domains={len(unique)} diversity={diversity...

🛠️ Reader to Writer Bridge

1. Read: explain score linking-domain diversity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 380.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 382 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'links={total} unique domains={len(unique)} diversity={diversity}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score linking-domain diversity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The size of the domain set over the total link count is a diversity ratio: 1.0 means every link is from a different site, lower means concentration.

```
backlinks = [
    'https://a.com/1', 'https://a.com/2', 'https://a.com/3',
    'https://b.com/x', 'https://b.com/y',
    'https://c.com/z',
]
def domain(u):
    return u.split('://', 1)[1].split('/', 1)[0]
unique = {domain(u) for u in backlinks}
total = len(backlinks)
diversity = round(len(unique) / total, 2)
print(f'links={total} unique domains={len(unique)} diversity={diversity}')
```

```
output
links=6 unique domains=3 diversity=0.5
```

How it works:

1. backlinks is a flat list of six full URLs, several sharing the same host.
2. domain splits off the scheme and path to leave the host string.
3. unique is the set of those hosts, here three, while total is six.
4. diversity is len(unique) over total, rounded to 0.5, printed beside the raw counts.

Try it:

1. Print the most over-represented domain alongside the score
2. Weight the score so brand-new domains count extra
3. Compare the diversity scores of two separate backlink lists
4. Add three more domains and watch diversity rise toward 1.0.

⚠ Common Mistakes

- Dividing total links by unique domains inverts the ratio into links-per-domain.
- Building the set from full URLs not hosts treats every page as its own domain.
- Rounding to zero decimals hides meaningful differences between profiles.

Skills gained `diversity score` `sets` `ratio`

→ **Where You'll Use This:** Diversity scores let you compare two backlink profiles at a glance and justify why a campaign chasing many new domains beats one squeezing more links from existing partners.

Mental Model

A garden with one species repeated is monoculture; many species is biodiversity. The diversity ratio tells you which kind of backlink garden you grew.

Project 382 Track Link Velocity Month by Month

The Idea Behind It

Link velocity is the pace at which new backlinks arrive, and a sudden burst can read as a paid-link blast to a search engine. This project walks a month-keyed dict in sorted order, prints each month's new links with the signed delta from the prior month, and finishes with the monthly average. The deltas turn a flat table into a trend you can read.

What You'll Build & Learn

Chart new backlinks month over month with signed deltas, and report the average monthly rate.

- Link velocity as backlink arrival rate
- Computing month-over-month deltas
- Carrying a previous value through a loop

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 381: score linking-domain diversity. Project 383 changes the surface problem while preserving the underlying method.

progress check

Reading target: track link velocity month by month is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `monthly = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.

- Evidence: `print('avg/month:', avg)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`monthly = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`if prev is None:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('avg/month:', avg)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if prev is None:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Track Link Velocity Month by Month: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: monthly, months, sorted, prev, is. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: monthly = { | Core: if prev is None: | Evidence: print('avg/month:', avg)

🛠️ Reader to Writer Bridge

1. Read: explain track link velocity month by month in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 381.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 383 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('avg/month:', avg)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `monthly = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of track link velocity month by month and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Sorting the month keys and subtracting each month's count from the previous one yields the velocity delta, with the first month seeded as a baseline.

```
monthly = {
    '2026-01': 12,
    '2026-02': 18,
    '2026-03': 25,
    '2026-04': 22,
    '2026-05': 40,
}
months = sorted(monthly)
prev = None
for m in months:
    n = monthly[m]
    if prev is None:
        print(f'{m}: {n} new (--)')
    else:
        delta = n - monthly[prev]
        sign = '+' if delta >= 0 else ''
        print(f'{m}: {n} new ({sign}{delta})')
    prev = m
avg = round(sum(monthly.values()) / len(monthly), 1)
print('avg/month:', avg)
```

```
output
2026-01: 12 new (--)
2026-02: 18 new (+6)
2026-03: 25 new (+7)
2026-04: 22 new (-3)
2026-05: 40 new (+18)
avg/month: 23.4
```

How it works:

1. monthly maps each year-month string to a count of new backlinks.
2. Iterating sorted months, the first prints with a placeholder delta of double dashes.
3. For later months, delta is the difference from the prior month with a plus sign when non-negative.
4. After the loop, avg is the mean of all monthly counts, here 23.4, printed last.

Try it:

1. Flag any month whose delta exceeds twice the running average
2. Compute a three-month rolling average instead of a single mean
3. Print the percentage change each month rather than the absolute delta
4. Add a month with a huge jump and flag it as a suspicious spike.

⚠ Common Mistakes

- Iterating an unsorted dict computes deltas against the wrong previous month.

- Forgetting the `prev is None` guard crashes on the first month with no predecessor.
- Reporting the cumulative total as velocity hides the per-month rate of change.

Skills gained link velocity | deltas | time series

→ Where You'll Use This: Tracking velocity spikes helps you distinguish organic growth from a manipulated burst, and explains sudden ranking changes when a competitor buys links en masse.

Mental Model

It is a speedometer for backlinks. Steady acceleration looks earned; a sudden floor-the-pedal jump makes the algorithm suspect a shortcut.

Project 383 Flag Toxic Links by Spammy TLD

The Idea Behind It

Some top-level domains are so cheap and abuse-prone that links from them are a toxicity signal. This project extracts the TLD from each backlink host with a right-split and checks it against a blocklist of known-spammy extensions like xyz, tk, and gq. Each link prints as TOXIC or ok, giving a fast first-pass filter before deeper manual review.

What You'll Build & Learn

Flag backlinks from spammy top-level domains using a blocklist as a quick toxicity filter.

- Spammy TLDs as a toxicity signal
- Extracting a TLD with rsplit
- Set membership for blocklist checks

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 382: track link velocity month by month. Project 384 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag toxic links by spammy tld is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

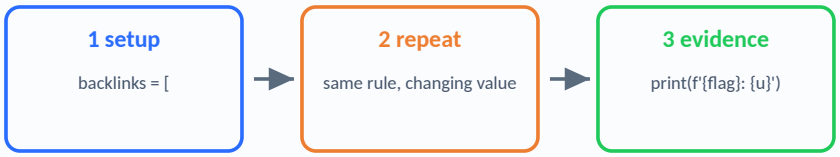
Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{flag}: {u}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`flag = 'TOXIC' if t in spammy else 'ok'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{flag}: {u}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = 'TOXIC' if t in spammy else 'ok'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Flag Toxic Links by Spammy TLD: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, https, shop, example, com. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: backlinks = [| Core: flag = 'TOXIC' if t in spammy else 'ok' | Evidence: print(f'{flag}: {u}')

🛠️ Reader to Writer Bridge

1. Read: explain flag toxic links by spammy tld in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 382.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 384 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{flag}: {u}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of flag toxic links by spammy tld and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Pulling the last dot-segment of each host yields the TLD, and membership in a spammy-TLD set flags the link as toxic.

```
backlinks = [
    'https://shop.example.com/a',
    'https://news.org/b',
    'https://spam.xyz/c',
    'https://casino.tk/d',
    'https://blog.com/e',
    'https://pills.gq/f',
]
spammy = {'xyz', 'tk', 'gq', 'ml', 'cf'}
def tld(u):
    host = u.split('://', 1)[1].split('/', 1)[0]
    return host.rsplit('.', 1)[-1]
for u in backlinks:
    t = tld(u)
    flag = 'TOXIC' if t in spammy else 'ok'
    print(f'{flag}: {u}')
```

```
output
ok: https://shop.example.com/a
ok: https://news.org/b
TOXIC: https://spam.xyz/c
TOXIC: https://casino.tk/d
ok: https://blog.com/e
TOXIC: https://pills.gq/f
```

How it works:

1. backlinks is a list of six URLs, some on suspicious extensions like xyz, tk, and gq.
2. spammy holds the blocklisted TLDs as a set for fast lookup.
3. tld splits off the scheme and path, then rsplits the host on the dot to grab the last label.
4. Each link prints TOXIC when its TLD is in the blocklist, otherwise ok.

Try it:

1. Add a few more abuse-prone TLDs you have seen in spam
2. Treat a link as toxic only if its TLD is spammy and its DR is low
3. Count how many toxic links each spammy TLD contributed
4. Add `.work` to the spammy set and re-run.

⚠ Common Mistakes

- Splitting on the first dot grabs a subdomain instead of the TLD.
- Comparing the whole host to the blocklist never matches a real domain.
- Treating a legitimate site that merely sits on a .xyz domain as automatically toxic.

Skills gained toxic links | TLD parsing | blocklist

→ Where You'll Use This: Before building a disavow file you triage thousands of backlinks, and a TLD blocklist quickly surfaces the obvious junk so manual review focuses on borderline cases.

⚡ Mental Model

Certain neighbourhoods of the web rent space to anyone for pennies. A link from one is like a reference written on the back of a betting slip.

Project 384 Compute the Deep-Link Ratio

💡 The Idea Behind It

A natural backlink profile points at inner pages, not just the homepage. This project counts how many backlink targets are the root path versus deeper URLs, then divides the deep count by the total to produce a deep-link ratio. A balanced result suggests organic interest in specific content, while an all-homepage profile can look artificially built.

What You'll Build & Learn

Compute the share of backlinks that point at inner pages rather than the homepage.

- Deep links versus homepage links
- Deep-link ratio as a naturalness gauge
- Counting by comparing each path to root

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 383: flag toxic links by spammy tld. Project 385 changes the surface problem while preserving the underlying method.

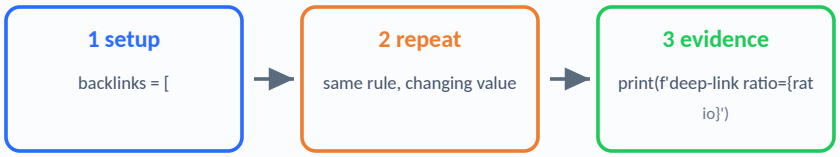
progress check

Reading target: compute the deep-link ratio is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'deep-link ratio={ratio}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`home = sum(1 for p in backlinks if p == '/')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'deep-link ratio={ratio}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `home = sum(1 for p in backlinks if p == '/')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute the Deep-Link Ratio: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, blog, post, products, shoes. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: backlinks = [| Core: home = sum(1 for p in backlinks if p == '/') | Evidence: print(f'deep-link ratio= {ratio}')

🛠️ Reader to Writer Bridge

1. Read: explain compute the deep-link ratio in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 383.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 385 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'deep-link ratio= {ratio}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of compute the deep-link ratio and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Counting links whose target is exactly `/` gives the homepage share, and the rest are deep links into inner pages, whose ratio reflects topical link spread.

```
backlinks = [
    '/',
    '/blog/post-1',
    '/',
    '/products/shoes',
    '/about',
    '/',
]
home = sum(1 for p in backlinks if p == '/')
deep = len(backlinks) - home
ratio = round(deep / len(backlinks), 2)
print(f'homepage links={home} deep links={deep}')
print(f'deep-link ratio={ratio}')
```

```
output
homepage links=3 deep links=3
deep-link ratio=0.5
```

How it works:

1. backlinks is a list of target paths, three of which are the bare root slash.
2. home counts how many entries equal the root path using a generator sum.
3. deep is simply the total minus the homepage count.
4. ratio is deep over the total, rounded to 0.5, printed beneath the two raw counts.

Try it:

1. Group deep links by their top-level path segment like blog or products
2. Flag the profile when the homepage share exceeds sixty percent
3. Handle target paths that include query strings or fragments
4. Add two more inner-page links and watch the ratio rise.

⚠ Common Mistakes

- Treating `/` and an empty path differently miscounts the homepage links.
- Counting `/index.html` as deep when it is really the homepage.
- Dividing by the homepage count instead of the total skews the ratio.

Skills gained deep-link ratio path matching naturalness

→ **Where You'll Use This:** A skewed deep-link ratio prompts content-led link building that earns references to specific articles, which both diversifies the profile and pushes equity to pages that convert.

⚡ Mental Model

If everyone only ever recommends the front door of a building, no one knows what is inside. Deep links are recommendations for individual rooms.

Project 385 Detect Reciprocal Links Between Sites

💡 The Idea Behind It

Two sites linking to each other carry less editorial weight than a one-way link, so spotting reciprocal pairs matters. This project walks the link graph and, for each edge, checks whether the target links back to the source. Matches are stored as sorted tuples in a set so each pair counts once regardless of direction, revealing possible link-exchange schemes.

What You'll Build & Learn

Detect mutual link pairs in a link graph and count them, surfacing possible exchange schemes.

- Reciprocal links carry less weight
- Checking if a target links back
- Sorted tuples in a set to dedupe pairs

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 384: compute the deep-link ratio. Project 386 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect reciprocal links between sites is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.

- Evidence: `print('count:', len(recip))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`if src in links.get(t, []):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('count:', len(recip))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if src in links.get(t, []):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Reciprocal Links Between Sites: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links, siteA, siteB, siteC, siteD. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: links = { | Core: if src in links.get(t, []): | Evidence: print('count:', len(recip))

🛠️ Reader to Writer Bridge

1. Read: explain detect reciprocal links between sites in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 384.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 386 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('count:', len(recip))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect reciprocal links between sites and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

For each edge A to B, checking whether B also links back to A finds the reciprocal pair, stored as a sorted tuple so each pair is reported once.

```
links = {
    'siteA': ['siteB', 'siteC'],
    'siteB': ['siteA'],
    'siteC': ['siteD'],
    'siteD': ['siteC'],
}
recip = set()
for src, outs in links.items():
    for t in outs:
        if src in links.get(t, []):
            recip.add(tuple(sorted((src, t))))
for pair in sorted(recip):
    print(f'reciprocal: {pair[0]} <-> {pair[1]}')
print('count:', len(recip))
```

```
output
reciprocal: siteA <-> siteB
reciprocal: siteC <-> siteD
count: 2
```

How it works:

1. links maps four sites to their outbound targets, with two mutual pairs hidden inside.
2. For each source and target, the code checks whether src appears in links.get(t, []).
3. A match is added as tuple(sorted src, t) so direction never duplicates the pair.
4. Sorted pairs print with an arrow, and the final count reports two reciprocal links.

Try it:

1. Report what fraction of all edges are reciprocal
2. Allow a site to link itself without falsely flagging a pair
3. Extend detection to three-site link rings, not just two-way pairs
4. Break one back-link and confirm that pair drops out.

⚠ Common Mistakes

- Storing pairs unsorted records `(A,B)` and `(B,A)` as two distinct reciprocals.
- Using `links[t]` instead of `links.get(t, [])` raises if a target has no entry.
- Counting both directions doubles the reciprocal-pair total.

Skills gained reciprocal links | graph traversal | sets

→ Where You'll Use This: Auditing a profile for excessive reciprocity catches link-exchange patterns that search engines discount, and steers outreach toward earning unreciprocated editorial links instead.

Mental Model

Two people praising each other on cue sounds rehearsed. A reciprocal link is that mutual back-pat, worth less than an unsolicited compliment.

Project 386 Apply Google's First-Link-Counts Rule

The Idea Behind It

When a page links to the same target twice, classic PageRank credits only the anchor text of the first link. This project models that first-link-counts rule by walking links in order and recording each target only the first time it appears. The result changes which anchor a target page is associated with, a subtlety that affects internal-link planning.

What You'll Build & Learn

Model the first-link-counts rule so each target keeps only its first anchor on a page.

- First link counts for anchor credit
- Recording a target only on first sight
- Order matters when iterating links

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 385: detect reciprocal links between sites. Project 387 changes the surface problem while preserving the underlying method.

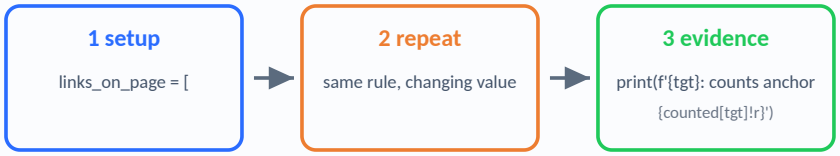
progress check

Reading target: apply google's first-link-counts rule is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `links_on_page = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{tgt}: counts anchor {counted[tgt]!r}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links_on_page = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`if link['to'] not in counted:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{tgt}: counts anchor {counted[tgt]!r}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if link['to'] not in counted:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Apply Google's First-Link-Counts Rule: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links_on_page, to, shoes, anchor, running. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: links_on_page = [| Core: if link['to'] not in counted: | Evidence: print(f'{tgt}: counts anchor {counted[tgt]!r}')

🛠️ Reader to Writer Bridge

1. Read: explain apply google's first-link-counts rule in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 385.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 387 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{tgt}: counts anchor {counted[tgt]!r}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `links_on_page = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of apply google's first-link-counts rule and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Walking the page's links in order and keeping only the first anchor seen per target mirrors the first-link-counts behaviour: later links to the same URL are ignored for anchor purposes.

```
links_on_page = [
    {'to': '/shoes', 'anchor': 'running shoes'},
    {'to': '/shoes', 'anchor': 'click here'},
    {'to': '/boots', 'anchor': 'boots'},
    {'to': '/boots', 'anchor': 'winter boots'},
]
counted = {}
```

```
for link in links_on_page:
    if link['to'] not in counted:
        counted[link['to']] = link['anchor']
for tgt in sorted(counted):
    print(f'{tgt}: counts anchor {counted[tgt]!r}')
```

```
output
/boots: counts anchor 'boots'
/shoes: counts anchor 'running shoes'
```

How it works:

1. links_on_page lists four links in order, with shoes and boots each appearing twice.
2. counted is a dict that records a target's anchor only if the target is not already a key.
3. Because iteration is ordered, the first link to each target wins its anchor slot.
4. Sorted output shows shoes credited to running shoes and boots to boots, the first anchors.

Try it:

1. Switch the order of the duplicate links and see the credited anchor change
2. Record all anchors per target but mark which one is credited
3. Count how many links were ignored because the target was already seen
4. Reorder the two `/shoes` links and watch the counted anchor change.

⚠ Common Mistakes

- Overwriting the stored anchor on every link keeps the last, not the first.
- Using a plain dict assignment without the `not in` guard loses the first-link rule.
- Assuming both links to a target pass anchor equity when only the first does here.

Skills gained first link counts | ordered dict | anchor credit

→ Where You'll Use This: When a navigation and an in-body link both point to the same page, knowing which anchor counts shapes how you place keyword-rich links so the right phrase is credited.

⚡ Mental Model

If you introduce a friend twice at a party, only the first introduction sticks. The first-link rule remembers the first name you gave them.

Project 387 Split Outbound Links Into Internal and External

💡 The Idea Behind It

A page's outbound links split into internal ones that recycle equity and external ones that leak it off-site. This project parses the host from each URL and partitions the link list by comparing that host to the site's own. Subdomains count as external here, which matters because the split is the foundation of both equity sculpting and outbound-link audits.

What You'll Build & Learn

Partition a page's outbound links into internal and external sets by comparing hosts.

- Internal links recycle equity, external leak it
- Comparing each link's host to the site
- Subdomains treated as external here

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 386: apply google's first-link-counts rule. Project 388 changes the surface problem while preserving the underlying method.

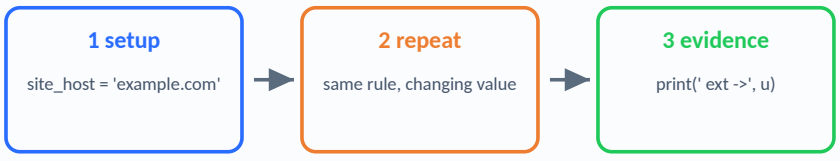
progress check

Reading target: split outbound links into internal and external is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `site_host = 'example.com'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(' ext ->', u)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`site_host = 'example.com'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`internal = [u for u in links if host(u) == site_host]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(' ext ->', u)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `internal = [u for u in links if host(u) == site_host]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Split Outbound Links Into Internal and External: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: site_host, example, com, links, https. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: site_host = 'example.com' | Core: internal = [u for u in links if host(u) == site_host] | Evidence: print(' ext ->', u)

🛠️ Reader to Writer Bridge

1. Read: explain split outbound links into internal and external in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 386.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 388 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' ext ->', u)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `site_host = 'example.com'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split outbound links into internal and external and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Comparing each link's host against the site's own host partitions the links into internal and external sets, the first step of any link-flow analysis.

```
site_host = 'example.com'
links = [
    'https://example.com/about',
    'https://blog.example.com/x',
    'https://other.com/ref',
    'https://example.com/shop',
    'https://partner.org/y',
]
def host(u):
    return u.split('://', 1)[1].split('/', 1)[0]
internal = [u for u in links if host(u) == site_host]
external = [u for u in links if host(u) != site_host]
print('internal:', len(internal))
print('external:', len(external))
for u in sorted(external):
    print('  ext ->', u)
```

```
output
```

```
internal: 2
external: 3
  ext -> https://blog.example.com/x
  ext -> https://other.com/ref
  ext -> https://partner.org/y
```

How it works:

1. site_host is example.com and links is a list of five outbound URLs.
2. host splits off the scheme then the path to isolate each URL's host.
3. internal keeps links whose host equals site_host; external keeps the rest.
4. Counts print, and each external link is listed sorted, including the blog subdomain.

Try it:

1. Treat subdomains of the site as internal rather than external
2. Count the equity each external link would receive given a page rank
3. Report the external domains, deduplicated, instead of full URLs
4. Decide subdomains are internal and adjust the host comparison.

⚠ Common Mistakes

- Treating a subdomain like `blog.example.com` as internal when it is a separate host here.
- Matching on substring rather than exact host counts `notexample.com` as internal.
- Forgetting to sort the external list makes the output nondeterministic.

Skills gained internal external | host parsing | equity sculpting

→ Where You'll Use This: Outbound-link audits use this split to find pages leaking equity to low-value external sites, and to confirm internal links keep authority circulating within the domain.

Mental Model

Internal links are money spent inside your own shop; external links are money handed to neighbours. The split tells you where your equity goes.

Project 388 Compute an Authority Score From a Profile

The Idea Behind It

A profile is many numbers, but a single comparable score helps prioritise. This toy authority score weights the two things that matter most: distinct referring domains and dofollow links that pass equity. It multiplies the domain count by five and the dofollow count by two, then sums them, collapsing a whole profile into one figure you can track over time.

What You'll Build & Learn

Blend referring domains and dofollow links into one weighted authority score for a profile.

- Weighting domains above raw links
- Counting dofollow links in a profile
- Collapsing metrics into one score

🧭 Skill Roadmap

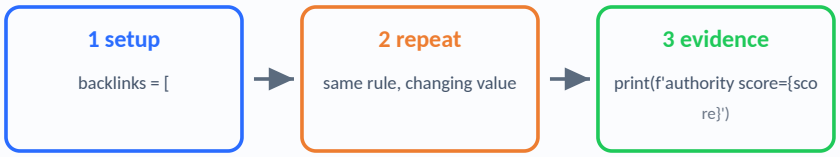
Skill step: moves you from recognition toward independent tracing. It builds on Project 387: split outbound links into internal and external. Project 389 changes the surface problem while preserving the underlying method. *progress check*

Reading target: compute an authority score from a profile is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'authority score={score}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus:

Move	Code to inspect	Why it matters
		model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'authority score={score}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute an Authority Score From a Profile: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, domain, com, rel, dofollow. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: backlinks = [| Core: dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow') | Evidence: print(f'authority score={score}')

Reader to Writer Bridge

1. Read: explain compute an authority score from a profile in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 387.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 389 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'authority score={score}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute an authority score from a profile and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Weighting unique referring domains more heavily than individual dofollow links and summing gives a single authority figure you can track over time.

```
backlinks = [
    {'domain': 'a.com', 'rel': 'dofollow'},
    {'domain': 'b.com', 'rel': 'dofollow'},
    {'domain': 'c.com', 'rel': 'nofollow'},
    {'domain': 'd.com', 'rel': 'dofollow'},
    {'domain': 'a.com', 'rel': 'nofollow'},
]
ref_domains = {b['domain'] for b in backlinks}
dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')
score = round(len(ref_domains) * 5 + dofollow * 2, 1)
print(f'ref domains={len(ref_domains)} dofollow links={dofollow}')
print(f'authority score={score}')
```

```
output
ref domains=4 dofollow links=3
authority score=26
```

How it works:

1. backlinks is a list of dicts each with a domain and a rel value.
2. ref_domains is the set of distinct domains, here four despite five links.

3. dofollow sums the links whose rel is dofollow, giving three.
4. score is four times five plus three times two, equal to 26, printed with its components.

Try it:

1. Subtract points for any toxic-TLD domains in the profile
2. Weight each domain by its DR before summing
3. Normalise the score to a zero-to-hundred scale
4. Add a link from a brand-new domain and watch the score jump.

⚠ Common Mistakes

- Counting total links instead of unique domains lets one spammy site inflate the score.
- Weighting dofollow and domains equally undervalues domain diversity.
- Including nofollow links in the dofollow term overstates passed equity.

Skills gained authority score | weighting | sets

→ Where You'll Use This: A blended score lets you rank prospect sites or track your own profile's growth month to month without staring at a dozen separate metrics each time.

Mental Model

Like a credit score, it boils many signals into one number. The weights decide which behaviours move the needle most.

Project 389 Apply a Disavow List to a Backlink Profile

The Idea Behind It

A disavow file tells Google to ignore specific toxic domains. This project applies that file locally by extracting each backlink's host and partitioning the list into kept and removed based on a disavow set. Recomputing your reported profile against only the kept links shows what your backlinks actually count for once the junk is stripped out.

What You'll Build & Learn

Apply a disavow set to a backlink list, separating kept links from disavowed ones.

- Disavow files ignore named domains
- Partitioning links by host membership
- Re-scoring the surviving profile

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 388: compute an authority score from a profile. Project 390 changes the surface problem while preserving the underlying method. *progress check*

Reading target: apply a disavow list to a backlink profile is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(' removed', u)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`kept = [u for u in backlinks if host(u) not in disavow]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(' removed', u)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `kept = [u for u in backlinks if host(u) not in disavow]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Apply a Disavow List to a Backlink Profile: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, https, good, com, spam. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: backlinks = [| Core: kept = [u for u in backlinks if host(u) not in disavow] | Evidence: print(' removed', u)

Reader to Writer Bridge

1. Read: explain apply a disavow list to a backlink profile in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 388.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 390 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' removed', u)`. Then compare punctuation, spacing, and order.
2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of apply a disavow list to a backlink profile and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

prediction before proof

same idea, new surface

🔍 From Concept to Code (Python)

Filtering the backlinks by whether each host appears in the disavow set partitions the profile into kept and disavowed links.

```
backlinks = [
    'https://good.com/a',
    'https://spam.xyz/b',
    'https://news.org/c',
    'https://badnet.tk/d',
    'https://blog.com/e',
]
disavow = {'spam.xyz', 'badnet.tk'}
def host(u):
    return u.split('://', 1)[1].split('/', 1)[0]
kept = [u for u in backlinks if host(u) not in disavow]
removed = [u for u in backlinks if host(u) in disavow]
print('kept:', len(kept))
print('disavowed:', len(removed))
for u in sorted(removed):
    print('  removed', u)
```

```
output
kept: 3
disavowed: 2
  removed https://badnet.tk/d
  removed https://spam.xyz/b
```

How it works:

1. backlinks is five full URLs and disavow is a set of two toxic hosts.
2. host strips the scheme and path to return each URL's domain.
3. kept keeps links whose host is not in disavow; removed keeps the rest.
4. Counts print and each removed link is listed sorted, confirming three kept and two dropped.

Try it:

1. Support disavowing an entire TLD, not just exact domains
2. Report the percentage of links removed by the disavow
3. Recompute the authority score on the kept links only
4. Add `blog.com` to the disavow set and watch a clean link get removed.

⚠ Common Mistakes

- Matching the full URL against the disavow set, which lists domains, never removes anything.
- Disavowing a domain but leaving its subdomains linked lets equity leak through.
- Forgetting to sort the removed list makes the audit output nondeterministic.

Skills gained disavow | filtering | host parsing

→ Where You'll Use This: After identifying toxic backlinks you build a disavow file for Search Console, and modelling its effect locally previews how much of your reported authority will remain.

⚡ Mental Model

A disavow file is a do-not-count list you hand the referee. These witnesses still spoke, but you have asked the judge to disregard them.

Project 390 Sculpt Equity by Nofollowing a Link

💡 The Idea Behind It

PageRank sculpting once meant adding nofollow to low-value links so more equity reached the rest. This project models that by storing each link as a target-and-rel tuple, then computing shares only over the followed links. With one of home's three links nofollowed, the remaining two each get half the equity instead of a third, demonstrating the redistribution.

What You'll Build & Learn

Show how nofollowing one link reshapes the equity share each followed link receives.

- Nofollow excludes a link from equity flow
- Sculpting concentrates equity on followed links
- Counting only followed links for shares

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 389: apply a disavow list to a backlink profile. Project 391 changes the surface problem while preserving the underlying method.

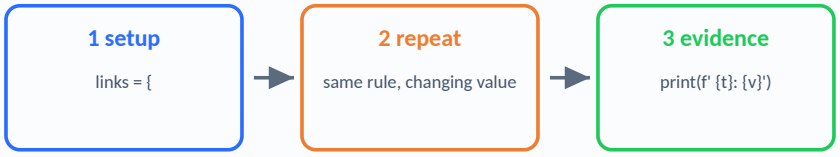
progress check

Reading target: sculpt equity by nofollowing a link is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f' {t}: {v}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`followed = [t for t, rel in outs if rel == 'follow']`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f' {t}: {v}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `followed = [t for t, rel in outs if rel == 'follow']`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Sculpt Equity by Nofollowing a Link: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links, home, follow, promo, nofollow. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: links = { | Core: followed = [t for t, rel in outs if rel == 'follow'] | Evidence: print(f' {t}: {v}')

🛠️ Reader to Writer Bridge

1. Read: explain sculpt equity by nofollowing a link in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 389.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 391 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f' {t}: {v}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sculpt equity by nofollowing a link and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Counting only followed out-links as the divisor means a nofollowed link no longer dilutes the share handed to the followed ones.

```
links = {
    'home': [('a', 'follow'), ('b', 'follow'), ('promo', 'nofollow')],
    'a': [('home', 'follow')],
    'b': [('home', 'follow')],
    'promo': [('home', 'follow')],
}
equity = {}
for src, outs in links.items():
    followed = [t for t, rel in outs if rel == 'follow']
    n = len(followed)
    share = round(1.0 / n, 3) if n else 0.0
    for t in followed:
        equity[(src, t)] = share
print('home passes equity to:')
for (src, t), v in sorted(equity.items()):
    if src == 'home':
        print(f'  {t}: {v}')
```

```
output
home passes equity to:
  a: 0.5
  b: 0.5
```

How it works:

1. links maps each page to a list of target-and-rel tuples, with home's promo link nofollowed.
2. For each source, followed filters to targets whose rel is follow.
3. share is one over the number of followed links, so home divides among two not three.
4. Equity entries print for home, showing a and b each receiving 0.5.

Try it:

1. Nofollow a second link and watch the remaining shares rise again
2. Treat nofollow as still consuming a share, the post-2009 behaviour
3. Report the equity each page sends across all sources, not just home
4. Remove the nofollow on `promo` and watch every share shrink.

⚠ Common Mistakes

- Dividing by all out-links including nofollow undoes the sculpting effect.
- A page whose every link is nofollow has no followed divisor and must be guarded.
- Believing modern Google still hoards the nofollowed link's share, which it no longer does.

Skills gained PageRank sculpting | nofollow | tuples

→ Where You'll Use This: Although modern engines treat nofollow as a hint, understanding sculpting clarifies historical advice and informs how you handle low-value links like login or cart pages today.

⚡ Mental Model

Sculpting is like sealing some of a watering can's spouts so the open ones gush harder. Each remaining flower drinks more.

Project 391 Find Pages With No Inbound Internal Links

💡 The Idea Behind It

An orphan page has no internal link pointing at it, so it receives almost no equity and may never get indexed. This project starts every page at zero inbound, walks the internal link graph to tally arrivals, then lists pages that stayed at zero. Surfacing these orphans is a routine step toward a fully crawlable site.

What You'll Build & Learn

Find orphan pages that receive no inbound internal links by listing those with zero in-degree.

- Orphan pages get no internal equity
- Initialising every page at zero inbound
- Listing pages with zero in-degree

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 390: sculpt equity by nofollowing a link. Project 392 changes the surface problem while preserving the underlying method.

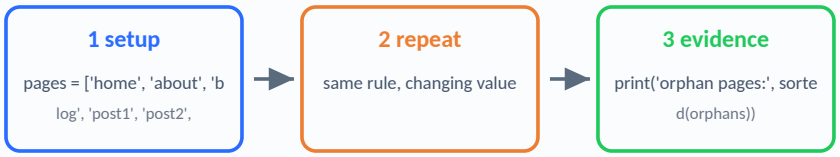
progress check

Reading target: find pages with no inbound internal links is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `pages = ['home', 'about', 'blog', 'post1', 'post2', 'contact']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print('orphan pages:', sorted(orphans))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = ['home', 'about', 'blog', 'post1', 'post2', 'contact']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`orphans = [p for p in pages if inbound[p] == 0]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('orphan pages:', sorted(orphans))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `orphans = [p for p in pages if inbound[p] == 0]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Pages With No Inbound Internal Links: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: pages, home, about, blog, post1. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: pages = ['home', 'about', 'blog', 'post1', 'post2', 'contact'] | Core: orphans = [p for p in pages if inbound[p] == 0] | Evidence: print('orphan pages:', sorted(orphans))

🛠️ Reader to Writer Bridge

1. Read: explain find pages with no inbound internal links in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 390.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 392 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('orphan pages:', sorted(orphans))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `pages = ['home', 'about', 'blog', 'post1', 'post2', 'contact']` and say why this line, value, import, or entry point is needed before the rest of the Python

can work.

3. Transfer: change one small part of find pages with no inbound internal links and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Initialising every known page at zero inbound, tallying targets, then keeping pages still at zero finds the orphans even though they never appear as a link target.

```
pages = ['home', 'about', 'blog', 'post1', 'post2', 'contact']
internal_links = {
    'home': ['about', 'blog'],
    'about': ['home'],
    'blog': ['post1'],
    'post1': ['blog'],
    'post2': ['blog'],
    'contact': [],
}
inbound = {p: 0 for p in pages}
for outs in internal_links.values():
    for t in outs:
        inbound[t] += 1
orphans = [p for p in pages if inbound[p] == 0]
print('orphan pages:', sorted(orphans))
```

```
output
orphan pages: ['contact', 'post2']
```

How it works:

1. pages lists all six known URLs, and internal_links maps each to its outbound targets.
2. inbound starts every page at zero, then nested loops increment each target's count.
3. orphans is a comprehension keeping pages whose inbound count is still zero.
4. Sorted orphans print, surfacing contact and post2 as unreached pages.

Try it:

1. Suggest a sensible parent page that could link to each orphan
2. Treat pages reachable only from the sitemap as still orphaned
3. Count near-orphans with exactly one inbound link as well
4. Link `home` to `contact` and watch it leave the orphan list.

⚠ Common Mistakes

- Building the inbound count only from targets misses orphans entirely (they never appear).
- Confusing pages with no out-links for orphans, which is the opposite direction.
- Treating the homepage as an orphan when external links or nav usually reach it.

Skills gained orphan pages | in-degree | crawlability

→ Where You'll Use This: Orphan detection is a standard technical-SEO finding, and fixing it by adding internal links from relevant pages restores both crawl access and equity flow to stranded content.

Mental Model

An orphan page is a room with no doors. The crawler can only find rooms it can walk into, so a doorless room stays invisible.

Project 392 Rank the Top Internally Linked Pages

The Idea Behind It

Sorting pages by inbound internal links reveals where your own site concentrates equity. This project tallies in-degree with a Counter, then sorts pages by descending count with a secondary alphabetical tiebreak before printing the top three. Whether the leaders match your intended priority pages is the test of an intentional internal-linking structure.

What You'll Build & Learn

Rank the most internally linked pages so you can confirm equity flows to your priority URLs.

- Ranking pages by inbound internal links
- Descending sort with alphabetical tiebreak
- Slicing the top results

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 391: find pages with no inbound internal links. Project 393 changes the surface problem while preserving the underlying method.

progress check

Reading target: rank the top internally linked pages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{p}: {indeg[p]} inbound')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`for outs in links.values():`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{p}: {indeg[p]} inbound')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for outs in links.values():`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Rank the Top Internally Linked Pages: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, links, home, blog. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import Counter | Core: for outs in links.values(): | Evidence: print(f'{p}: {indeg[p]} inbound')

🛠️ Reader to Writer Bridge

1. Read: explain rank the top internally linked pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 391.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 393 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{p}: {indeg[p]} inbound')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of rank the top internally linked pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

After tallying in-degree, sorting by negative count with the page name as a tiebreaker gives a deterministic top-N of the most internally linked pages.

```
from collections import Counter

links = {
    'home': ['blog', 'shop', 'about'],
    'blog': ['shop', 'post1'],
    'about': ['shop'],
    'post1': ['shop', 'blog'],
    'shop': [],
}
indeg = Counter()
for outs in links.values():
    for t in outs:
        indeg[t] += 1
ranked = sorted(indeg, key=lambda p: (-indeg[p], p))
for p in ranked[:3]:
    print(f'{p}: {indeg[p]} inbound')
```

```
output
shop: 4 inbound
blog: 2 inbound
about: 1 inbound
```

How it works:

1. links maps each page to its outbound targets, with shop appearing as a target repeatedly.
2. A Counter named indeg accumulates inbound counts across every target list.
3. sorted ranks pages by negative in-degree then name, so ties break alphabetically.
4. The top three slice prints shop, blog, and about with their inbound totals.

Try it:

1. Print all pages, not just the top three
2. Break ties by out-degree instead of alphabetical order
3. Compare the ranking against a hand-listed set of priority pages
4. Add two links to `post1` and watch it climb the ranking.

⚠ Common Mistakes

- Sorting by `indeg[p]` ascending lists the least-linked pages first.
- Omitting the page-name tiebreaker makes equal-count ties order unpredictably.
- Slicing before sorting returns an arbitrary subset, not the true top three.

Skills gained internal ranking | Counter | sorting

→ Where You'll Use This: This ranking validates an internal-linking strategy: if a thin page tops the list while a conversion page languishes, you rebalance links to send equity where revenue lives.

⚡ Mental Model

It is a leaderboard of which pages your site cheers loudest for. The names at the top should be the ones you actually want winning.

Project 393 Score Anchor-Text Diversity

💡 The Idea Behind It

A natural backlink profile spreads across many different anchors, while a manipulated one repeats the same phrase. This project rewards that variety by combining two factors: the ratio of unique anchors to total, and one minus the dominant anchor's share. Multiplying them yields a score that drops when either uniqueness is low or one anchor dominates.

What You'll Build & Learn

Score anchor-text diversity by combining unique-anchor share with how much the top anchor dominates.

- Diversity rewards many distinct anchors
- Penalising a single dominant anchor
- Combining two factors into one score

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 392: rank the top internally linked pages. Project 394 changes the surface problem while preserving the underlying method.

progress check

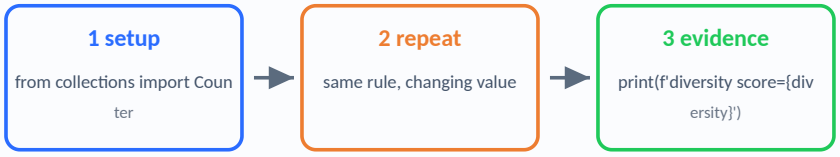
Reading target: score anchor-text diversity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'diversity score={diversity}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`counts = Counter(a.lower() for a in anchors)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'diversity score={diversity}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `counts = Counter(a.lower() for a in anchors)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score Anchor-Text Diversity: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, anchors, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: counts = Counter(a.lower() for a in anchors) | Evidence: print(f'diversity score={diversity}')

🛠️ Reader to Writer Bridge

1. Read: explain score anchor-text diversity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 392.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 394 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'diversity score={diversity}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of score anchor-text diversity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Multiplying the unique-anchor ratio by `1 minus the top anchor's share` gives a score that drops both when anchors repeat and when one phrase dominates.

```
from collections import Counter

anchors = ['running shoes', 'click here', 'Acme', 'running shoes', 'shoes',
'read more']
counts = Counter(a.lower() for a in anchors)
total = len(anchors)
unique = len(counts)
top_share = max(counts.values()) / total
diversity = round(unique / total * (1 - top_share), 3)
print(f'unique anchors={unique}/{total}')
print(f'top anchor share={round(top_share, 3)}')
print(f'diversity score={diversity}')
```

```
output
unique anchors=5/6
top anchor share=0.333
diversity score=0.556
```

How it works:

1. anchors is six entries, with running shoes appearing twice after lowercasing.
2. Counter gives per-anchor counts, unique is the distinct count, total is six.
3. top_share is the most common anchor's count over total, here a third.
4. diversity multiplies unique over total by one minus top_share, printing 0.556.

Try it:

1. Weight branded anchors so they boost the diversity score
2. Use the count distribution's entropy as an alternative score
3. Report which anchor is the dominant one alongside the score
4. Make every anchor identical and watch the score collapse to zero.

⚠ Common Mistakes

- Using unique-over-total alone ignores one anchor that dominates the rest.
- Counting anchors case-sensitively splits `Shoes` and `shoes` into two.
- Dividing by unique anchors rather than total inverts the ratio.

Skills gained anchor diversity scoring Counter

→ Where You'll Use This: A diversity score lets you compare anchor profiles across pages or against competitors, and flags when a link campaign is leaning too hard on one keyword phrase.

Mental Model

A choir of many voices sounds rich; one voice drowning the rest sounds forced. The score rewards the choir and docks the soloist.

Project 394 Find Backlinks Pointing at a 404

The Idea Behind It

A backlink pointing at a page that now returns 404 wastes its equity completely. This project cross-checks each backlink's recorded status code, filters to those that are not 200, and lists the broken ones with their source and target. The wasted-equity total it reports is reclaimable: a redirect on each dead target recovers the lost link power.

What You'll Build & Learn

Find backlinks whose targets return a 404 so their wasted equity can be reclaimed with redirects.

- Backlinks to 404 pages waste equity
- Filtering records by status code
- Equity is reclaimable via redirects

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 393: score anchor-text diversity. Project 395 changes the surface problem while preserving the underlying method. *progress check*

Reading target: find backlinks pointing at a 404 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.

- Evidence: `print('total wasted:', len(broken))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`broken = [b for b in backlinks if b['status'] != 200]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('total wasted:', len(broken))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `broken = [b for b in backlinks if b['status'] != 200]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Backlinks Pointing at a 404: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, https, com, to, shoes. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: backlinks = [| Core: broken = [b for b in backlinks if b['status'] != 200] | Evidence: print('total wasted:', len(broken))

🛠️ Reader to Writer Bridge

1. Read: explain find backlinks pointing at a 404 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 393.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 395 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('total wasted:', len(broken))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find backlinks pointing at a 404 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering the backlinks for any non-200 target status surfaces the broken ones whose passed equity is currently lost.

```
backlinks = [
    {'from': 'https://a.com', 'to': '/shoes', 'status': 200},
    {'from': 'https://b.com', 'to': '/old-page', 'status': 404},
    {'from': 'https://c.com', 'to': '/boots', 'status': 200},
    {'from': 'https://d.com', 'to': '/gone', 'status': 404},
]
broken = [b for b in backlinks if b['status'] != 200]
print('broken backlinks (link equity wasted):')
for b in sorted(broken, key=lambda x: x['from']):
    print(f"  {b['from']} -> {b['to']} ({b['status']})")
print('total wasted:', len(broken))
```

```
output
broken backlinks (link equity wasted):
  https://b.com -> /old-page (404)
  https://d.com -> /gone (404)
total wasted: 2
```

How it works:

1. backlinks is a list of dicts each holding from, to, and a status code.
2. broken is a comprehension keeping backlinks whose status is not 200.
3. The broken links print sorted by source URL with their target and status.
4. A final line reports two wasted backlinks pointing at 404 pages.

Try it:

1. Include 410 and 500 statuses as wasted alongside 404
2. Suggest a redirect target for each broken backlink
3. Sum the wasted equity weighted by each source's authority
4. Add a 301 target and decide whether it should count as broken.

⚠ Common Mistakes

- Checking only for 404 misses 410 and 5xx targets that also waste the link.
- Treating a 301 target as broken when it actually still forwards the equity.
- Sorting by status instead of source URL makes the report order unstable.

Skills gained broken backlinks | status codes | filtering

→ Where You'll Use This: Finding backlinks to dead pages is a high-value quick win: a single redirect recovers authority that has been sitting unused, often lifting rankings with no new outreach.

Mental Model

A backlink to a 404 is a cheque written to a closed account. The money is real, but it bounces until you reopen the account with a redirect.

Project 395 Find the Link Gap Versus a Competitor

The Idea Behind It

A link-gap analysis lists the domains linking to a competitor but not to you, the most actionable backlink-prospecting output there is. This project models both profiles as sets of domains and uses set subtraction to compute the gap in a single operation. The two domains it surfaces are concrete outreach targets that already link to similar content.

What You'll Build & Learn

Compute the link gap by subtracting your referring domains from a competitor's set.

- Link-gap analysis finds prospect domains
- Set subtraction for the difference
- Gap domains are outreach targets

Skill Roadmap

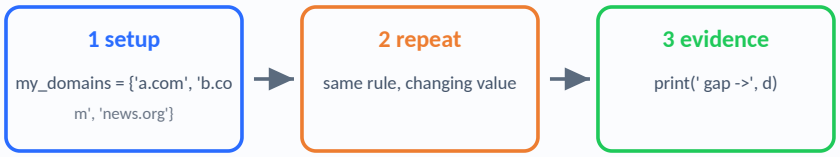
Skill step: pushes the same idea into a less automatic situation. It builds on Project 394: find backlinks pointing at a 404. Project 396 changes the surface problem while preserving the underlying method. *progress check*

Reading target: find the link gap versus a competitor is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `my_domains = {'a.com', 'b.com', 'news.org'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(' gap ->', d)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`my_domains = {'a.com', 'b.com', 'news.org'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`for d in sorted(gap):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(' gap ->', d)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for d in sorted(gap):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find the Link Gap Versus a Competitor: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: my_domains, com, news, org, competitor_domains. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: my_domains = {'a.com', 'b.com', 'news.org'} | Core: for d in sorted(gap): | Evidence: print(' gap ->', d)

🛠️ Reader to Writer Bridge

1. Read: explain find the link gap versus a competitor in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 394.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 396 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' gap ->', d)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `my_domains = {'a.com', 'b.com', 'news.org'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find the link gap versus a competitor and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Subtracting your referring-domain set from the competitor's reveals exactly the domains you are missing and could pitch for a link.

```
my_domains = {'a.com', 'b.com', 'news.org'}
competitor_domains = {'a.com', 'news.org', 'forbes.com', 'wired.com', 'b.com'}
gap = competitor_domains - my_domains
print('competitor backlink domains:', len(competitor_domains))
print('domains we are missing:', len(gap))
```

```
    for d in sorted(gap):
        print('  gap ->', d)
```

```
output
competitor backlink domains: 5
domains we are missing: 2
  gap -> forbes.com
  gap -> wired.com
```

How it works:

1. my_domains and competitor_domains are two sets of referring hosts.
2. gap is competitor_domains minus my_domains, the domains they have that you lack.
3. The competitor's total and the gap size print as counts.
4. Each gap domain lists sorted, surfacing forbes.com and wired.com as prospects.

Try it:

1. Compute the gap against several competitors at once and union the results
2. Report the overlap of domains linking to both you and the rival
3. Rank gap domains by an attached DR value before listing
4. Add a domain you both share and confirm it is not in the gap.

⚠ Common Mistakes

- Subtracting in the wrong order returns domains only you have, not the gap.
- Comparing lists instead of sets breaks the subtraction and keeps duplicates.
- Forgetting to sort the gap makes the prospecting list nondeterministic.

Skills gained link gap | set difference | prospecting

→ Where You'll Use This: Link-gap reports drive real outreach plans: a site already linking your rival is far likelier to link you, making the gap list the warmest set of prospects available.

Mental Model

It is a guest list comparison: who came to your rival's party but skipped yours. Those are precisely the people worth inviting next.

Project 396 Band Referring Domains by Domain Rating

The Idea Behind It

Linking domains are not equal; a DR 80 site dwarfs a DR 15 one. This project bands each referring domain into high, mid, or low tiers using simple threshold rules in a band function, then tallies the bands with a Counter.

The resulting shape, two high and two low here, describes the quality profile rather than just the quantity.

What You'll Build & Learn

Band referring domains into high, mid, and low tiers by domain rating to show profile quality.

- Domain rating reflects link value
- Banding values with threshold rules
- Tallying bands with Counter

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 395: find the link gap versus a competitor. Project 397 changes the surface problem while preserving the underlying method.

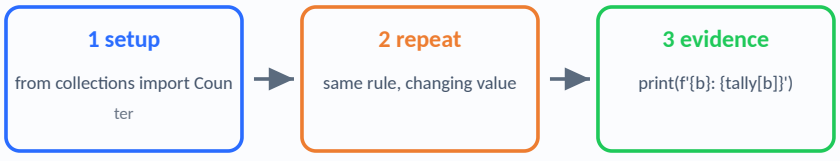
progress check

Reading target: band referring domains by domain rating is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{b}: {tally[b]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`if dr >= 70:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{b}: {tally[b]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if dr >= 70:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Band Referring Domains by Domain Rating: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, backlinks, domain, com. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: if dr >= 70: | Evidence: print(f'{b}: {tally[b]}')

🛠️ Reader to Writer Bridge

1. Read: explain band referring domains by domain rating in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 395.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 397 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{b}: {tally[b]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of band referring domains by domain rating and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A small banding function buckets each domain rating into high, mid, or low, and a Counter tallies how the profile's authority is distributed.

```
from collections import Counter

backlinks = [
    {'domain': 'a.com', 'dr': 80},
    {'domain': 'b.com', 'dr': 30},
    {'domain': 'c.com', 'dr': 75},
    {'domain': 'd.com', 'dr': 15},
    {'domain': 'e.com', 'dr': 60},
]
def band(dr):
    if dr >= 70:
        return 'high'
    if dr >= 40:
        return 'mid'
    return 'low'
tally = Counter(band(b['dr']) for b in backlinks)
for b in ('high', 'mid', 'low'):
    print(f'{b}: {tally[b]}')
```

```
output
high: 2
mid: 1
low: 2
```

How it works:

1. backlinks is a list of dicts each with a domain and a dr rating.
2. band returns high at 70 and above, mid at 40 and above, otherwise low.
3. Counter tallies the band of every backlink's rating.
4. The three bands print in a fixed high-mid-low order with their counts.

Try it:

1. Add a fourth elite band for DR 90 and above
2. Weight each band when summing into an overall score
3. Print the actual domains within each band, not just counts
4. Move a band edge to 50 and watch a domain change bands.

⚠ Common Mistakes

- Off-by-one band edges (using `>` instead of `>=`) misfile a domain on the boundary.
- Iterating the Counter for output gives nondeterministic band order.
- Averaging the ratings hides a profile that is all high or all low.

Skills gained domain rating | banding | Counter

→ Where You'll Use This: Banding by DR shows whether your authority comes from a few strong domains or a long tail of weak ones, guiding whether to chase quality or breadth next.

⚡ Mental Model

Sorting links by DR is like grading references by who wrote them. A note from a CEO outweighs ten from strangers, so you band them accordingly.

Project 397 Express PageRank as a Share of Equity

💡 The Idea Behind It

Raw PageRank values are abstract decimals that mean little to a stakeholder. This project runs the full damped PageRank to convergence, then divides each page's rank by the total and multiplies by a hundred to express its equity as a percentage. The same converged numbers become a share-of-equity split that a non-technical audience can immediately grasp.

What You'll Build & Learn

Convert converged PageRank values into percentage shares of total equity for easy communication.

- Raw PageRank is hard to interpret
- Normalising rank into a percentage
- Communicating equity to stakeholders

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 396: band referring domains by domain rating. Project 398 changes the surface problem while preserving the underlying method.

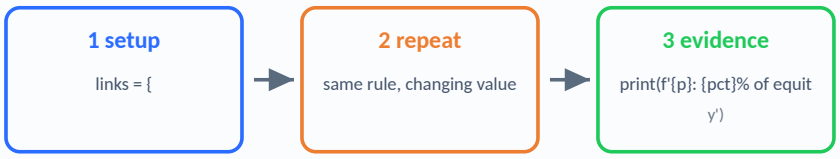
progress check

Reading target: express pagerank as a share of equity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `links = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f'{p}: {pct}% of equity')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`links = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus:

Move	Code to inspect	Why it matters
		model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`rank = {p: 1.0 / N for p in links}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{p}: {pct}% of equity')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rank = {p: 1.0 / N for p in links}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Express PageRank as a Share of Equity: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: links, home, len, rank, range. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: links = { | Core: rank = {p: 1.0 / N for p in links} | Evidence: print(f'{p}: {pct}% of equity')

Reader to Writer Bridge

1. Read: explain express pagerank as a share of equity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 396.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 398 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{p}: {pct}% of equity')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `links = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of express pagerank as a share of equity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Running the damped power method to convergence then dividing each page's rank by the total turns the stationary vector into readable percentage shares.

```
links = {
    'home': ['a', 'b'],
    'a': ['c'],
    'b': ['c'],
    'c': ['home'],
}
N = len(links)
d = 0.85
rank = {p: 1.0 / N for p in links}
for _ in range(50):
    nxt = {p: (1 - d) / N for p in links}
    for p, outs in links.items():
        share = rank[p] / len(outs)
        for t in outs:
            nxt[t] += d * share
    rank = nxt
total = sum(rank.values())
for p in sorted(rank):
    pct = round(100 * rank[p] / total, 1)
    print(f'{p}: {pct}% of equity')
```

```
output
a: 17.4% of equity
b: 17.4% of equity
c: 33.3% of equity
home: 32.0% of equity
```

How it works:

1. The familiar four-page graph runs damped PageRank for fifty iterations.
2. total sums all converged ranks, expected to be close to one.
3. For each page, pct is 100 times its rank over total, rounded to one place.
4. Output reads as percentages, with c holding 33.3 percent of the equity.

Try it:

1. Render the percentages as a simple text bar chart
2. Print the page holding the largest equity share explicitly
3. Show the percentage change after adding one new internal link
4. Add a link into `a` and watch its equity share rise.

⚠ Common Mistakes

- Dividing by `N` instead of the rank total gives the wrong percentage when ranks vary.
- Reading raw ranks as percentages already, which only holds if they sum to exactly 100.
- Too few iterations reports shares before the ranks have settled.

Skills gained PageRank normalisation reporting

→ Where You'll Use This: Presenting internal authority as percentage shares makes link-structure recommendations land with clients and executives who would glaze over at raw 0.33 figures.

Mental Model

Saying a page has 0.33 PageRank means nothing; saying it holds a third of the site's authority pie makes the slice instantly clear.

Project 398 Pick the Strongest Backlink

The Idea Behind It

When reporting wins or prioritising outreach, the single best backlink is a dofollow link from the highest-authority domain. This project finds it with one max call over a tuple key that ranks dofollow status first, then DR, then domain name for a stable tiebreak. The tuple ordering encodes the priority logic in a single readable line.

What You'll Build & Learn

Pick the strongest backlink using a tuple sort that prioritises dofollow then domain rating.

- Strongest link is dofollow and high DR
- Tuple keys for multi-level sorting
- Booleans sort after numbers in a tuple

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 397: express pagerank as a share of equity. Project 399 changes the surface problem while preserving the underlying method.

progress check

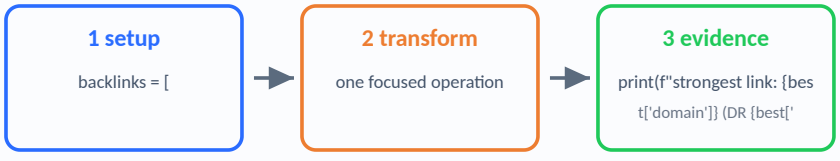
Reading target: pick the strongest backlink is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is deliberately small: Python stores a value, applies one clear operation, and prints evidence. That tiny chain is the reading pattern larger programs still use.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f"strongest link: {best['domain']} (DR {best['dr']}, {best['rel']})")` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`best = max(backlinks, key=lambda b: (b['rel'] == 'dofollow', b['dr'], b['domain']))`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print(f"strongest link: {best['domain']} (DR {best['dr']}, {best['rel']})")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `best = max(backlinks, key=lambda b: (b['rel'] == 'dofollow', b['dr'], b['domain']))`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Pick the Strongest Backlink: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, domain, com, dr, rel. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: backlinks = [| Core: best = max(backlinks, key=lambda b: (b['rel'] == 'dofollow', b['dr'], b... | Evidence: print(f"strongest link: {best['domain']} (DR {best['dr']}, {best['rel']}...

🛠️ Reader to Writer Bridge

1. Read: explain pick the strongest backlink in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 397.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 399 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"strongest link: {best['domain']} (DR {best['dr']}, {best['rel']})")`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of pick the strongest backlink and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `max` keyed on a tuple of `(is dofollow, domain rating, domain)` ranks equity-passing links first, then by rating, with the name as a deterministic tiebreaker.

```
backlinks = [
    {'domain': 'a.com', 'dr': 80, 'rel': 'dofollow'},
    {'domain': 'b.com', 'dr': 20, 'rel': 'nofollow'},
    {'domain': 'c.com', 'dr': 50, 'rel': 'dofollow'},
    {'domain': 'd.com', 'dr': 90, 'rel': 'dofollow'},
]
best = max(backlinks, key=lambda b: (b['rel'] == 'dofollow', b['dr'],
b['domain']))
print(f"strongest link: {best['domain']} (DR {best['dr']}, {best['rel']})")
```

```
output
strongest link: d.com (DR 90, dofollow)
```

How it works:

1. backlinks is four dicts each carrying a domain, a dr, and a rel value.
2. max uses a tuple key of dofollow truth, then dr, then domain name.
3. Python compares the tuples left to right, so dofollow status dominates the choice.
4. The winner d.com prints with its DR 90 and dofollow status.

Try it:

1. Return the top three strongest links instead of just one
2. Penalise toxic-TLD domains so they never win
3. Break DR ties by the freshest link date
4. Make `d.com` nofollow and watch the winner change to `c.com`.

⚠ Common Mistakes

- Keying on domain rating alone could crown a high-DR nofollow link that passes no equity.
- Omitting the domain tiebreaker makes equal-rated links resolve unpredictably.
- Comparing the boolean after the rating lets a strong nofollow link beat a weak dofollow one.

Skills gained best backlink | tuple sort | max

→ Where You'll Use This: Highlighting the single strongest backlink gives reports a clear headline win, and the same priority logic ranks outreach prospects when you can only pursue a few.

⚡ Mental Model

It is picking the trophy reference from a stack: first toss the ones that do not count, then take the most prestigious name remaining.

Project 399 Measure a New Link's PageRank Lift

💡 The Idea Behind It

To prove an internal-linking change actually moved equity, you compute PageRank before and after the change and compare. This project wraps PageRank in a reusable function, runs it on a base graph and a boosted graph that adds one link, then prints each page's rank as a before-and-after delta. The shift in b's rank is the equity the new link moved.

What You'll Build & Learn

Quantify a new link's impact by comparing PageRank before and after the link is added.

- Refactoring PageRank into a function
- A B comparison of two link graphs

- Reading the delta as moved equity

Skill Roadmap

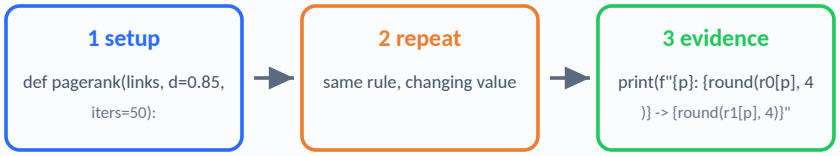
Skill step: pushes the same idea into a less automatic situation. It builds on Project 398: pick the strongest backlink. Project 400 changes the surface problem while preserving the underlying method. *progress check*

Reading target: measure a new link's pagerank lift is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def pagerank(links, d=0.85, iters=50):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: A nested loop means outer choice plus inner repeated work. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print(f"{p}: {round(r0[p], 4)} -> {round(r1[p], 4)}")` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def pagerank(links, d=0.85, iters=50):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`share = rank[p] / len(outs) if outs else 0.0`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f"{p}: {round(r0[p], 4)} -> {round(r1[p], 4)}")`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `share = rank[p] / len(outs) if outs else 0.0`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Measure a New Link's PageRank Lift: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: pagerank, links, iters, len, rank. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def pagerank(links, d=0.85, iters=50): | Core: share = rank[p] / len(outs) if outs else 0.0 | Evidence: print(f"{p}: {round(r0[p], 4)} -> {round(r1[p], 4)}")

Reader to Writer Bridge

1. Read: explain measure a new link's pagerank lift in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 398.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 400 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f"{p}: {round(r0[p], 4)} -> {round(r1[p], 4)}")`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `def pagerank(links, d=0.85, iters=50):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure a new link's pagerank lift and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Wrapping the damped power method in a function lets you run it on the base and boosted graphs and print each page's before-to-after rank shift.

```
def pagerank(links, d=0.85, iters=50):
    N = len(links)
    rank = {p: 1.0 / N for p in links}
    for _ in range(iters):
        nxt = {p: (1 - d) / N for p in links}
        for p, outs in links.items():
            share = rank[p] / len(outs) if outs else 0.0
            for t in outs:
                nxt[t] += d * share
        rank = nxt
    return rank

base = {'home': ['a'], 'a': ['home'], 'b': ['home']}
boosted = {'home': ['a'], 'a': ['home', 'b'], 'b': ['home']}
r0 = pagerank(base)
r1 = pagerank(boosted)
for p in sorted(base):
    print(f"{p}: {round(r0[p], 4)} -> {round(r1[p], 4)}")
```

```
output
a: 0.4636 -> 0.3878
b: 0.05 -> 0.2148
home: 0.4864 -> 0.3974
```

How it works:

1. pagerank is a reusable function returning converged ranks for any links dict.
2. base and boosted differ by one link: a now also points at b in boosted.
3. r0 and r1 hold the converged ranks for the two graphs.
4. Each page prints its before-and-after value, with b rising from 0.05 to 0.2148.

Try it:

1. Add several links at once and attribute the total lift
2. Report each page's percentage change rather than the raw delta
3. Test removing a link and confirm the equity flows back
4. Add the new link from `b` instead and compare which placement lifts `b` more.

⚠ Common Mistakes

- Reusing one `rank` dict across both runs leaks the first result into the second.
- Comparing ranks from different iteration counts is not a fair before-and-after.
- Forgetting the dangling guard `if outs` crashes when a node has no out-links.

Skills gained before after | PageRank function | deltas

→ Where You'll Use This: Modelling a link change before shipping it lets you predict and then prove its effect, turning internal-linking work from guesswork into a measured, defensible recommendation.

Mental Model

It is a before-and-after photo for link equity. Lay the two PageRank snapshots side by side and the difference is the change you made.

Project 400 Judge Anchor-Profile Health

The Idea Behind It

Two signals together flag an unnatural backlink profile: an exact-match anchor share that is too high and too few distinct anchors. This project computes both, the exact-match percentage and the unique-anchor ratio, and collects any triggered warnings into a flags list. A non-empty list yields a RISKY verdict with reasons, while a clean one reads HEALTHY.

What You'll Build & Learn

Combine exact-match share and anchor uniqueness into one healthy-or-risky profile verdict.

- Two signals combine for a verdict
- Collecting reasons in a flags list
- Building a verdict string from flags

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 399: measure a new link's pagerank lift. Project 401 changes the surface problem while preserving the underlying method.

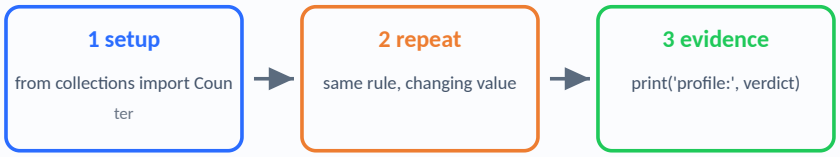
progress check

Reading target: judge anchor-profile health is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print('profile:', verdict)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`if exact_pct > 30:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('profile:', verdict)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if exact_pct > 30:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Judge Anchor-Profile Health: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, anchors, running, shoes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: from collections import Counter | Core: if exact_pct > 30: | Evidence: print('profile:', verdict)

🛠️ Reader to Writer Bridge

1. Read: explain judge anchor-profile health in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 399.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 401 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('profile:', verdict)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of judge anchor-profile health and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Computing the exact-match percentage and the unique-anchor ratio and testing each against a threshold collects the failing signals into a risk verdict.

```
from collections import Counter

anchors = ['running shoes', 'running shoes', 'running shoes', 'running shoes',
           'Acme', 'click here', 'shop now', 'best shoes']
target = 'running shoes'
total = len(anchors)
exact = Counter(a.lower() for a in anchors)[target]
exact_pct = round(100 * exact / total, 1)
unique = len(set(a.lower() for a in anchors))
flags = []
if exact_pct > 30:
    flags.append('exact-match over-optimised')
if unique / total < 0.5:
    flags.append('low diversity')
verdict = 'HEALTHY' if not flags else 'RISKY: ' + ', '.join(flags)
print(f'exact={exact_pct}% unique={unique}/{total}')
print('profile:', verdict)
```

```
output
exact=50.0% unique=5/8
profile: RISKY: exact-match over-optimised
```

How it works:

1. anchors stacks four exact-match phrases against four varied ones, total eight.
2. exact_pct is the exact-match share and unique is the count of distinct lowercased anchors.
3. Flags append when exact share tops 30 percent or uniqueness falls below half.
4. Because exact is 50 percent, the verdict reads RISKY with the over-optimised reason.

Try it:

1. Add a third flag for an excessive naked-url share
2. Return a numeric risk score alongside the verdict text
3. Make each threshold a tunable parameter of a function

4. Add four distinct anchors and watch the profile flip to HEALTHY.

⚠ Common Mistakes

- Reporting only the exact-match share misses a profile that repeats a branded anchor.
- Using `and` instead of collecting independent flags hides one problem behind the other.
- Counting anchors case-sensitively splits identical phrases and overstates diversity.

Skills gained profile health | multi-signal | verdict

→ Where You'll Use This: A combined health check is what an audit tool reports for a profile, letting you summarise multiple risk factors in one verdict a stakeholder can act on.

⚡ Mental Model

It is a medical checkup reading two vitals at once. Either reading alone might pass, but both off together earns a clear diagnosis.

Project 401 Summarise a Backlink Profile's Vital Signs

💡 The Idea Behind It

One pass over a backlink fixture should yield the numbers an SEO checks first. This project computes them together: total links, distinct referring domains via a set, dofollow count, toxic links flagged by spammy TLD, and deep links that avoid the homepage. Bundling these vital signs into one readout gives the snapshot you would open a backlink review with.

What You'll Build & Learn

Compute a backlink profile's headline vital signs in a single pass over the fixture.

- Headline backlink metrics in one pass
- Combining set, sum, and TLD checks
- Building a profile snapshot

🧭 Skill Roadmap

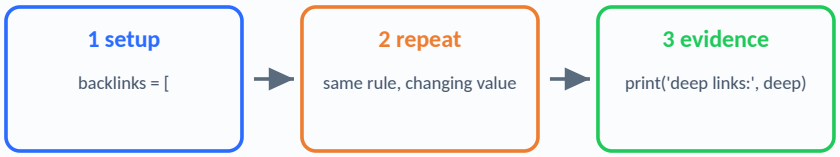
Skill step: pushes the same idea into a less automatic situation. It builds on Project 400: judge anchor-profile health. Project 402 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise a backlink profile's vital signs is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `backlinks = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print('deep links:', deep)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`backlinks = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('deep links:', deep)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Summarise a Backlink Profile's Vital Signs: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: backlinks, domain, com, dr, rel. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: backlinks = [| Core: dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow') | Evidence: print('deep links:', deep)

🛠️ Reader to Writer Bridge

1. Read: explain summarise a backlink profile's vital signs in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 400.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 402 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('deep links:', deep)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `backlinks = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of summarise a backlink profile's vital signs and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A handful of comprehensions over the same backlink list extract each headline metric, turning a raw link table into a profile snapshot.

```
backlinks = [
    {'domain': 'a.com', 'dr': 80, 'rel': 'dofollow', 'to': '/shoes'},
    {'domain': 'b.com', 'dr': 20, 'rel': 'nofollow', 'to': '/'},
    {'domain': 'a.com', 'dr': 80, 'rel': 'dofollow', 'to': '/'},
    {'domain': 'spam.xyz', 'dr': 5, 'rel': 'dofollow', 'to': '/shoes'},
    {'domain': 'c.com', 'dr': 60, 'rel': 'dofollow', 'to': '/boots'},
]
spammy = {'xyz', 'tk', 'gq'}
domains = {b['domain'] for b in backlinks}
dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')
toxic = sum(1 for b in backlinks if b['domain'].rsplit('.', 1)[-1] in spammy)
deep = sum(1 for b in backlinks if b['to'] != '/')
print('total backlinks:', len(backlinks))
print('referring domains:', len(domains))
print('dofollow:', dofollow)
print('toxic links:', toxic)
print('deep links:', deep)
```

```
output
total backlinks: 5
referring domains: 4
dofollow: 4
toxic links: 1
deep links: 3
```

How it works:

1. backlinks is five dicts each with a domain, dr, rel, and to path.
2. domains is the set of distinct domains and dofollow sums the equity-passing links.
3. toxic counts links whose TLD is in the spammy set, and deep counts non-root targets.
4. Five labelled lines print total links, domains, dofollow, toxic, and deep counts.

Try it:

1. Add anchor diversity as a sixth vital sign
2. Return the metrics as a dict instead of printing them
3. Compute the dofollow share as a percentage of total

4. Add a backlink from a fifth domain and watch the domain count rise.

⚠ Common Mistakes

- Counting links where you meant referring domains conflates volume with reach.
- Reading the TLD with the wrong split flags clean domains as toxic.
- Treating an empty-path target as deep miscounts the deep-link total.

Skills gained profile snapshot | vital signs | aggregation

→ Where You'll Use This: These five numbers are the top of any backlink dashboard, so a one-pass summary like this is the natural first screen of a monitoring or reporting tool.

Mental Model

It is the patient chart at the foot of the bed: a few key vitals glanced at first to judge whether the profile is healthy.

Project 402 Run a Full Off-Page Authority Audit

The Idea Behind It

This capstone ties the chapter together by running every off-page check on a single backlink fixture. From one list of dicts it derives backlink and referring-domain counts, the dofollow split, toxic links by TLD, anchor diversity via a Counter, and a blended authority score that rewards domains and dofollow while subtracting heavily for toxic links. The printed audit reads like a real report.

What You'll Build & Learn

Run a complete off-page audit that reports every key backlink metric and a blended score at once.

- Combining every off-page metric in one audit
- Penalising toxic links in the score
- Anchor diversity via Counter

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 401: summarise a backlink profile's vital signs. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run a full off-page authority audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
- Evidence: `print('authority score:', score)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
2. Apply the rule	`dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('authority score:', score)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Off-Page, Links and Authority becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run a Full Off-Page Authority Audit: connect the surface task to the chapter skill, model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics.
Names	Watch these names first: collections, Counter, backlinks, domain, news. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow') | Evidence: print('authority score:', score)

🛠️ Reader to Writer Bridge

1. Read: explain run a full off-page authority audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 401.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('authority score:', score)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of run a full off-page authority audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Set, sum, and Counter operations over the same backlink records compute every off-page metric at once, then a weighted formula that penalises toxic links yields the headline score.

```
from collections import Counter

backlinks = [
    {'domain': 'news.org', 'dr': 75, 'rel': 'dofollow', 'to': '/shoes',
'anchor': 'running shoes'},
    {'domain': 'blog.com', 'dr': 40, 'rel': 'dofollow', 'to': '/', 'anchor':
'Acme'},
    {'domain': 'spam.xyz', 'dr': 5, 'rel': 'dofollow', 'to': '/shoes',
'anchor': 'cheap pills'},
    {'domain': 'forum.net', 'dr': 30, 'rel': 'nofollow', 'to': '/boots',
'anchor': 'click here'},
    {'domain': 'news.org', 'dr': 75, 'rel': 'dofollow', 'to': '/boots',
'anchor': 'winter boots'},
]
spammy = {'xyz', 'tk', 'gq'}
domains = {b['domain'] for b in backlinks}
dofollow = sum(1 for b in backlinks if b['rel'] == 'dofollow')
toxic = [b for b in backlinks if b['domain'].rsplit('.', 1)[-1] in spammy]
anchors = Counter(b['anchor'].lower() for b in backlinks)
score = len(domains) * 5 + dofollow * 2 - len(toxic) * 10
print('=== Off-Page Audit ===')
print('backlinks:', len(backlinks))
print('referring domains:', len(domains))
print('dofollow/nofollow:', f'{dofollow}/{len(backlinks) - dofollow}')
print('toxic links:', len(toxic))
print('anchor diversity:', f'{len(anchors)}/{len(backlinks)}')
print('authority score:', score)
```

```
output
=== Off-Page Audit ===
backlinks: 5
referring domains: 4
dofollow/nofollow: 4/1
toxic links: 1
anchor diversity: 5/5
authority score: 18
```

How it works:

1. backlinks is five dicts carrying domain, dr, rel, to path, and anchor.

2. domains, dofollow, and toxic are derived as sets and sums, with anchors counted.
3. score adds five per domain and two per dofollow, then subtracts ten per toxic link.
4. A titled block prints backlinks, domains, the dofollow split, toxic, diversity, and the score of 18.

Try it:

1. Add a deep-link ratio line to the audit block
2. Flag the audit as risky when the score drops below zero
3. Export the full audit as a JSON object instead of printed lines
4. Disavow the `spam.xyz` link first and watch the score climb.

⚠ Common Mistakes

- Letting toxic links count toward authority instead of subtracting overstates the score.
- Counting links rather than unique domains lets one site dominate the score.
- Building the anchor Counter without lowercasing splits identical anchors.

Skills gained off-page audit | blended score | reporting

→ Where You'll Use This: A consolidated audit like this is the deliverable a client expects: every signal computed from one dataset and rolled into a single headline authority figure they can track.

Mental Model

It is the full physical, not a single vital. Every test from the chapter runs on one sample and the results print as one chart.

✅ Chapter Progress Checkpoint

What this chapter added: after 32 mini-projects, Off-Page, Links and Authority should feel like a reusable Python move: model internal link equity with a toy pagerank, anchor-text distribution and backlink-profile metrics. *chapter checkpoint*

1 Read
Open any project from Projects 371-402 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 12 (Local and Map SEO) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

9 questions · 4 categories · graded

2 Recall | 2 Meaning | 2 Structure | 3 Transfer

11.1.Q1 Transfer · MEDIUM

Turn back to Project 386 (Apply Google's First-Link-Counts Rule). Without running it, what exactly does it print?

11.1.Q2 Meaning · MEDIUM

Turn back to Project 381 (Score Linking-Domain Diversity). Without running it, what exactly does it print?

11.1.Q3 Structure · MEDIUM

Turn back to Project 398 (Pick the Strongest Backlink). Without running it, what exactly does it print?

11.1.Q4 Transfer · MEDIUM

Turn back to Project 388 (Compute an Authority Score From a Profile). Without running it, what exactly does it print?

11.1.Q5 Recall · EASY

Which backlink profile is strongest, all else equal?

- A. 100 links from 1 domain
- B. 50 links from 2 domains
- C. 20 links from 20 different domains
- D. 200 links from 1 domain

11.1.Q6 Meaning · MEDIUM

In the toy PageRank with damping d = 0.85, why does each iteration seed every page with (1 - d)/N before adding the link flow, and why do the final ranks sum to 1.0?

11.1.Q7 Structure · MEDIUM

A page has rank 0.4 and you increase its out-links from 2 to 4. What happens to the equity each out-link passes (even-split rule)?

- A. Stays 0.2 each
- B. Doubles to 0.4 each
- C. Halves from 0.2 to 0.1 each
- D. Goes to zero

11.1.Q8 Transfer · MEDIUM

Your monitoring shows 80% of a page's backlinks use the exact phrase "running shoes" as anchor text. Why is this a red flag rather than a win?

11.1.Q9 Recall EASY What is the difference between a dofollow and a nofollow link for ranking?

- A Nofollow links are faster to crawl
- B Only dofollow links pass equity (vote); nofollow signals interest but passes no PageRank
- C Nofollow links rank higher
- D They are identical

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 11 – Off-Page, Links and Authority

11.1.Q1. `/boots: counts anchor 'boots'` then `/shoes: counts anchor 'running shoes'`

11.1.Q2. `links=6 unique domains=3 diversity=0.5`

11.1.Q3. `strongest link: d.com (DR 90, dofollow)`

11.1.Q4. `ref domains=4 dofollow links=3` then `authority score=26`

11.1.Q5. C referring domains are independent votes; 20 distinct domains beat any number of links from just one or two sites.

11.1.Q6. The (1 - d)/N term is the 15% "teleport" baseline a random surfer keeps regardless of links, so no page starves; equity is only redistributed, never created or destroyed, so the total is conserved at 1.0.

11.1.Q7. C 0.4 split over 2 links is 0.2 each; split over 4 it is 0.1 each. More out-links dilute every link's share.

11.1.Q8. Natural profiles spread anchors across branded, generic, naked-URL and partial phrases. An 80% exact-match share looks bought/manipulated (over-optimized) and risks a penalty; a ~30% line typically separates natural from suspicious.

11.1.Q9. B dofollow links pass equity and count as votes; nofollow links may drive traffic but do not move PageRank.

Chapter 12 · Local and Map SEO

Check NAP consistency across citations, score local signals, and audit business-profile completeness.

Chapter Skill Map

What this chapter adds: In Local and Map SEO, the focus is check nap consistency across citations, score local signals, and audit business-profile completeness. You practise it across 30 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 12 assumes the reading habits from Chapter 11: Off-Page, Links and Authority. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 13, Log-File and Server Analysis, reuses this chapter's discipline with a new kind of problem.

1 **Projects 403-412**	2 **Projects 413-422**	3 **Projects 423-432**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Local ranking is three forces: relevance × distance × prominence, all resting on one foundation — your NAP being the *same* everywhere. *who you are, how near, how trusted*

When you search "plumber near me" Google does not show ten blue links first — it shows a local pack: a map and three business profiles. Getting into that pack is a different game from ranking a web page. Google answers one question: *of the verified businesses that could serve this searcher, which three are the best match?* It weighs three signals — relevance (does the business do what was asked), distance (how near is it to the searcher), and prominence (how well-known and trusted is it).

All three rest on a quieter foundation: NAP — your business **N**ame, **A**ddress, and **P**hone. Google sees your business not as one record but as dozens of mentions scattered across Yelp, Apple Maps, Facebook, directories, and your own site. These mentions are called citations. When they all state the *same* name, address, and phone, Google is confident they describe one real business and trusts it more. When they drift — "Acme Plumbing" here, "Acme Plumbing Co" there, two different phone numbers — confidence drops and so does ranking.

So the first job of local SEO is mechanical, not creative: prove that every citation says the same thing. The trouble is that humans type the same facts a dozen ways. Before you can compare two citations you must normalize them — strip case, whitespace, and noise down to a canonical form. A phone number is the cleanest example: (415) 555-0199 and +1 415-555-0199 are the same number, so reduce every variant to bare digits and compare those:

```
phones = ['(415) 555-0199', '415.555.0199', '+1 415-555-0199']
def digits(p):
```

```
    d = ''.join(c for c in p if c.isdigit())
    if len(d) == 11 and d.startswith('1'):
        d = d[1:]          # drop the US country code
    return d
norms = {digits(p) for p in phones}
print('distinct numbers:', len(norms))
```

```
output
distinct numbers: 1
```

Three spellings collapse to one number, so these citations agree. Run the same idea on names (drop "LLC", "Inc") and addresses ("Street" → "St", "North" → "N") and you can diff any two citations field by field. Consistency is something you measure, not something you eyeball — and once it is a number, you can score it, track it, and fix the worst offenders first. *normalize, then compare*

Relevance and prominence then build on top: relevance comes from picking the right business categories and a complete profile; prominence comes from reviews (count, rating, freshness) and from being cited across many kinds of sources. Distance you cannot change — but knowing it explains the pack you compete in.

NAP

Name, Address, Phone — the three core facts that identify a business. Local SEO hinges on these being byte-for-byte identical across every place the business is listed.

CITATION

Any online mention of a business's NAP — a Yelp page, an Apple Maps listing, a directory entry, a chamber-of-commerce page. The more citations agree, the more Google trusts the data.

NAP CONSISTENCY

The share of a business's citations whose NAP exactly matches the canonical record. A core local-SEO health metric: a high percentage signals one trustworthy business, a low one signals confusion.

LOCAL PACK (MAP PACK)

The block of (usually three) business profiles shown on a map at the top of a local-intent search. Appearing here generally requires a verified profile with a rating, reviews, and a category.

RELEVANCE / DISTANCE / PROMINENCE

Google's three stated local-ranking factors. Relevance = does the business match the query; Distance = how near it is to the searcher; Prominence = how well-known and reviewed it is.

GOOGLE BUSINESS PROFILE (GBP)

The business's listing on Google — name, address, phone, hours, categories, description, photos. Completeness and verification of this profile feed relevance and prominence.

Normalize before you compare

You cannot diff two citations until they are written the same way. The canonical form strips everything that is noise: leading/trailing spaces, case, and legal suffixes like "LLC" or "Inc". Only then does "Joe's Coffee House, LLC" line up with "joe's coffee house". This same normalize-then-compare pattern runs through the whole chapter — for names, phones, and addresses alike:

```
name = "  Joe's Coffee House, LLC  "
suffixes = (' llc', ' inc', ' inc.', ' co', ' co.', ' ltd')
norm = name.strip().lower()
for s in suffixes:
    if norm.endswith(s):
        norm = norm[:-len(s)].rstrip(' ,')
        break
print('normalized:', repr(norm))
```

```
output
normalized: "joe's coffee house"
```

Addresses differ only by abbreviation — "North Main Street" versus "N Main St" — so map words to canonical short forms and the two spellings compare equal. Normalization turns "looks the same to a human" into "is equal to a computer."

Score NAP consistency as one number

Per-source diffs tell you *which* field drifted on *which* directory. But you also want one headline figure: what share of your citations exactly match the canonical NAP. That percentage is the metric you report and track over time. Here five directories are checked against the canonical `(name, phone, zip)`:

```
canonical = ('acme plumbing', '4155550199', '94110')
citations = {
    'Yelp':       ('acme plumbing',    '4155550199', '94110'),
    'Bing':       ('acme plumbing',    '4155550199', '94110'),
    'Apple':      ('acme plumbing co', '4155550199', '94110'),  # name drift
```

```
    'Facebook':   ('acme plumbing',    '4155550100', '94110'),  # phone drift
    'Foursquare': ('acme plumbing',    '4155550199', '94110'),
}
consistent = sum(1 for c in citations.values() if c == canonical)
total = len(citations)
print('NAP consistency:', round(100 * consistent / total, 1), '%')
```

```
output
NAP consistency: 60.0 %
```

Three of five match, so consistency is 60%. The two offenders are not random: a stray "co" suffix and a transposed phone digit. Fixing those two listings is the single highest-leverage local-SEO action here — it lifts the headline number and removes the confusion that suppresses ranking.

Relevance: a complete, correctly-categorized profile

Relevance is "does this business do what was asked." Two things drive it: choosing the right `categories` (exactly one *primary*, the rest secondary, all genuinely on-topic) and filling out the profile. A profile with hours, a description, and photos signals a real, relevant business and outranks a bare one. Score completeness by counting filled fields:

```
profile = {'name': 'Acme Plumbing', 'hours': {'mon': '9–5'},
           'categories': ['Plumber'], 'photos': 6,
           'description': '', 'website': 'https://acme.example',
           'phone': '4155550199'}
fields = ['name', 'hours', 'categories', 'photos',
          'description', 'website', 'phone']
missing = [f for f in fields if not profile.get(f)]
print('missing:', missing)
print('completeness:', round(100 * (len(fields) - len(missing)) / len(fields), 1),
'%')
```

```
output
missing: ['description']
completeness: 85.7 %
```

One empty field — the description — caps this profile at 85.7%. A common trap is "improving" relevance by stuffing the *name* with a keyword ("San Francisco Plumber Co"). That is against Google's rules and risks suspension: your name field must be your real name, and an off-topic category like "Coffee Shop" on a plumber actively hurts you.

Prominence: reviews are the loudest signal

Prominence is "how well-known and trusted." For a local business the loudest prominence signal is reviews — and not just the average. Three numbers matter: the `count`, the `average rating`, and the `velocity` (a steady stream of fresh reviews signals an active business). Replying to reviews, especially negative ones, is itself a trust signal:

```
reviews = [5, 4, 5, 3, 5, 4, 2, 5, 4, 5]
n = len(reviews)
```

```
avg = sum(reviews) / n
print('count:', n, '| average:', round(avg, 2), '| stars:', round(avg))
```

```
output
count: 10 | average: 4.2 | stars: 4
```

Ten reviews averaging 4.2 round to a four-star snippet in results. But an average hides shape — a 4.2 made of all 4s and 5s is healthier than a 4.2 with a pile of 1s pulled up by 5s — so a real audit also builds a star-by-star histogram and watches for a *falling* average, which is the earliest warning of a problem.

Distance and the local pack

Distance you cannot edit, but you can compute it. With latitude and longitude you approximate kilometres by flat-earth scaling — degrees times 111 km, with longitude shrunk by the cosine of the latitude — and rank businesses by nearness to the searcher. To even *be* a candidate for the pack, a profile generally needs a rating, enough reviews, a category, and verification:

```
profile = {'rating': 4.5, 'reviews': 27,
           'category': 'Plumber', 'verified': True}
checks = {'has rating': profile['rating'] > 0,
          'has reviews': profile['reviews'] >= 5,
          'has category': bool(profile['category']),
          'verified': profile['verified']}
print('local-pack eligible:', all(checks.values()))
```

```
output
local-pack eligible: True
```

All four gates pass, so this profile is eligible. Real ranking then blends the signals at different weights — NAP consistency, reviews, proximity, completeness, categories — into one composite score. You cannot move distance, so you compete on the levers you *can* move: consistency, completeness, and reviews.

relevance and prominence are the levers; distance is the field

Example 1

Your Yelp listing shows phone "(415) 555-0199" and your Google profile stores "4155550199". A teammate says they mismatch. Are these citations consistent?

SOLUTION

1. Normalize both to bare digits — strip everything that is not a digit. Yelp: "(415) 555-0199" → `4155550199`.
2. Google already stores "4155550199" → `4155550199`. Neither has a leading "1" country code to drop.
3. Compare the canonical forms: 4155550199 == 4155550199.

They are consistent — the phones are identical once normalized.

formatting is not a mismatch; only the digits matter

Example 2

A business is checked against four citations. Three exactly match its canonical NAP and one has a different phone. The owner asks for a single health number. What is the NAP consistency, and what is the fix?

SOLUTION

1. Count the citations whose full NAP matches the canonical record: `3 of 4`.
2. NAP consistency = 100 × consistent / total = 100 × 3 / 4 = `75.0%`.
3. The one offender drifted on the *phone* field, so correct that listing's phone to the canonical number.

NAP consistency is 75.0%; fixing the single mismatched phone restores it toward 100%.

one wrong field on one source measurably lowers the headline metric

Example 3

A plumber's profile has a 4.5 rating, 27 reviews, a "Plumber" category, and is verified — but its name field reads "Cheap #1 Plumber Near Me". Is it local-pack eligible, and is anything wrong?

SOLUTION

1. Run the four eligibility gates: rating > 0 (yes), reviews ≥ 5 (yes), has a category (yes), verified (yes) — all pass, so it is `eligible`.
2. But the name embeds a keyword and a promotional symbol; the real name is not "Cheap #1 Plumber Near Me".
3. A keyword-stuffed name violates Google's naming rules and `risks suspension`, which would remove it from the pack entirely.

Eligible on the four gates, but the stuffed name is a suspension risk — fix the name to the real business name.

eligibility is necessary, not sufficient; rule violations can still pull you

Mental-Math Shortcut

Same facts everywhere, then earn the three R's. First make your Name, Address, Phone *identical* across every citation — normalize and diff, do not eyeball. Then work the levers you control: **R**elevance (right categories, complete profile), p**R**ominence (more, fresher, answered reviews). Distance is the one R you cannot move, so never let a fixable consistency or review problem be the reason you lose to a nearer rival.

⚠ Common Mistakes

- Eyeballing NAP instead of normalizing it — "(415) 555-0199" and "4155550199" look different but are the same number, while "Acme Plumbing" and "Acme Plumbing Co" look the same but are a real mismatch.
- Keyword-stuffing the business name ("San Francisco Plumber Co") or adding off-topic categories ("Coffee Shop" on a plumber) — both violate Google's rules and can get the profile suspended rather than boosted.

- Chasing the average star rating while ignoring review count, velocity, and replies — a stale 5.0 with three reviews loses to an active 4.4 with two hundred answered ones.
- Letting duplicate listings stand — two profiles for one business split the reviews and confuse Google; merge or remove duplicates detected by a normalized (name, phone) key.

This chapter contains 30 hands-on mini-projects (Projects 403–432). Type each one out, run it, and experiment.

Project 403 Normalize a Business Name for Matching

The Idea Behind It

Directories spell the same business a dozen ways, so before any comparison you fold a name to one canonical form. This code lowercases and strips the string, then walks a tuple of legal suffixes and, when the name ends with one, slices it off and trims trailing commas and spaces, turning Joe's Coffee House, LLC into a clean key.

What You'll Build & Learn

Reduce a messy business name to a canonical lowercase form with legal suffixes removed, ready for cross-directory matching.

- String stripping and lowercasing
- Matching with endswith against a tuple
- Slicing and rstrip to drop suffixes

Skill Roadmap

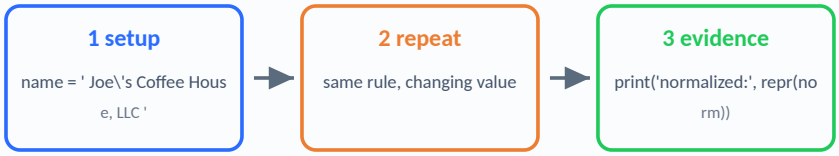
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 404 changes the surface problem while preserving the underlying method. *progress check*

Reading target: normalize a business name for matching is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `name = ' Joe\'s Coffee House, LLC '` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('normalized:', repr(norm))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`name = ' Joe\'s Coffee House, LLC '`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`if norm.endswith(s):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('normalized:', repr(norm))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if norm.endswith(s):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Normalize a Business Name for Matching: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: name, Joe, Coffee, House, suffixes. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: name = ' Joe\'s Coffee House, LLC ' | Core: if norm.endswith(s): | Evidence: print('normalized:', repr(norm))

🛠️ Reader to Writer Bridge

1. Read: explain normalize a business name for matching in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 404 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('normalized:', repr(norm))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `name = ' Joe\'s Coffee House, LLC '` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of normalize a business name for matching and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Lowercasing, trimming whitespace, and chopping a trailing legal suffix collapses cosmetic variants into one comparable key, which is the basis of every NAP match.

```
name = '  Joe\'s Coffee House, LLC  '
suffixes = (' llc', ' inc', ' inc.', ' co', ' co.', ' ltd')
norm = name.strip().lower()
for s in suffixes:
    if norm.endswith(s):
        norm = norm[:-len(s)].rstrip(' ,')
```

```
        break
print(repr(name.strip()))
print('normalized:', repr(norm))
```

```
output
"Joe's Coffee House, LLC"
normalized: "joe's coffee house"
```

How it works:

1. The raw name carries surrounding spaces, mixed case, and a trailing LLC suffix that must all go before comparison.
2. norm = name.strip().lower() removes outer whitespace and folds case into one baseline.
3. The for loop tests each suffix with endswith, and on a hit slices it away with norm[:-len(s)] then rstrips stray commas and spaces.
4. break stops after the first matching suffix, and the prints show the trimmed original beside the normalized key.

Try it:

1. Add more suffixes such as corp, llp, and pllc to the tuple and confirm each gets stripped.
2. Strip a registered-trademark or copyright symbol from the name before the suffix loop runs.
3. Collapse repeated internal spaces so double spaces between words become one.
4. Add ' corporation' to the suffix tuple and feed in 'Acme Corporation'.

⚠ Common Mistakes

- Checking suffixes before lowercasing means ' LLC' never matches the lowercase ' llc' list.
- Forgetting to `rstrip(' ,')` after slicing leaves a dangling comma like 'joe's coffee house,'.
- Not `break`-ing after the first match could strip two suffixes from a name that ends in two suffix-like words.

Skills gained string normalization | suffix stripping | canonicalization

→ Where You'll Use This: Every citation-cleanup tool starts here: you canonicalize business names so the same listing on Yelp, Bing, and Apple Maps collapses to a single entity you can audit.

Mental Model

Like filing every spelling of a customer's name under one index card so two paperwork piles about the same person finally meet.

Project 404 Normalize a Phone Number to Digits

The Idea Behind It

A single phone number wears many costumes: parentheses, dots, a country code, or bare digits. The digits function keeps only characters that pass isdigit, then drops a leading 1 from any eleven-digit result so a US country code does not split otherwise-identical numbers. Feeding every variant through a set proves they collapse to one distinct value.

What You'll Build & Learn

Turn formatted phone strings into bare ten-digit numbers and count how many distinct numbers a list really contains.

- Filtering characters with isdigit
- Stripping a leading country code
- Using a set to count distinct values

🧭 Skill Roadmap

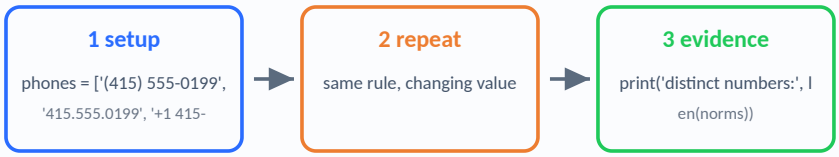
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 403: normalize a business name for matching. Project 405 changes the surface problem while preserving the underlying method. *progress check*

Reading target: normalize a phone number to digits is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `phones = ['(415) 555-0199', '415.555.0199', '+1 415-555-0199', '4155550199']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('distinct numbers:', len(norms))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`phones = ['(415) 555-0199', '415.555.0199', '+1 415-555-0199', '4155550199']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`d = ''.join(c for c in p if c.isdigit())`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('distinct numbers:', len(norms))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `d = ''.join(c for c in p if c.isdigit())`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Normalize a Phone Number to Digits: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: phones, digits, join, isdigit, len. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: phones = ['(415) 555-0199', '415.555.0199', '+1 415-555-0199', '4155550... | Core: d = ''.join(c for c in p if c.isdigit()) | Evidence: print('distinct numbers:', len(norms))

🛠️ Reader to Writer Bridge

1. Read: explain normalize a phone number to digits in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 403.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 405 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('distinct numbers:', len(norms))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `phones = ['(415) 555-0199', '415.555.0199', '+1 415-555-0199', '4155550199']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of normalize a phone number to digits and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Keeping only `isdigit()` characters and trimming a leading US '1' on 11-digit numbers maps every formatting style onto one canonical 10-digit string.

```
phones = ['(415) 555-0199', '415.555.0199', '+1 415-555-0199', '4155550199']
def digits(p):
    d = ''.join(c for c in p if c.isdigit())
    if len(d) == 11 and d.startswith('1'):
        d = d[1:]
    return d
```

```
    norms = {digits(p) for p in phones}
    for p in phones:
        print(f'{p:18} -> {digits(p)}')
    print('distinct numbers:', len(norms))
```

```
output
(415) 555-0199     -> 4155550199
415.555.0199       -> 4155550199
+1 415-555-0199    -> 4155550199
4155550199         -> 4155550199
distinct numbers: 1
```

How it works:

1. The phones list holds four formats of the same number, differing only in punctuation and a country code.
2. digits joins characters where c.isdigit is true, discarding parentheses, dots, plus signs, and spaces.
3. When the result is eleven digits starting with 1, d = d[1:] removes the US country code so it matches the ten-digit forms.
4. A set comprehension over all four strings yields norms, and len(norms) prints 1, confirming every variant is the same number.

Try it:

1. Handle a leading plus and country code longer than one digit, such as a UK or German number.
2. Validate that the cleaned result has exactly ten digits and flag anything shorter as malformed.
3. Format the normalized digits back into a readable parenthesized display string.
4. Add an extension like '4155550199 x12' and decide how to handle the extra digits.

⚠ Common Mistakes

- Stripping the '1' from any 11-digit string would corrupt a real number that legitimately has 11 digits but no country code.
- Using `str.replace` for each separator misses unexpected ones; filtering on `isdigit()` catches them all.
- Comparing the formatted strings instead of the digit forms reports false mismatches.

Skills gained phone normalization | set deduplication | f-string formatting

→ Where You'll Use This: Phone numbers are the P in NAP, and matching them across listings depends on this digit-only canonical form before you ever flag a mismatch in real local-SEO work.

Mental Model

Like wiping the wrapper off a candy so you can tell whether two pieces from different shops are actually the same flavor.

Project 405 Normalize a Street Address

The Idea Behind It

Two addresses can mean one place yet read differently because of abbreviations. The code lowercases the address, replaces commas with spaces, and splits into words, then maps each word through a dictionary that turns street into st, avenue into ave, north into n, and suite into ste. Words absent from the map pass through unchanged via the dict default.

What You'll Build & Learn

Canonicalize a street address by abbreviating common words so two spellings of the same location compare equal.

- Tokenizing a string into words
- Dictionary get with a fallback default
- List comprehension to rewrite tokens

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 404: normalize a phone number to digits. Project 406 changes the surface problem while preserving the underlying method.

progress check

Reading target: normalize a street address is the surface task; the transferable Python move is to mark setup, rule, and output before running.

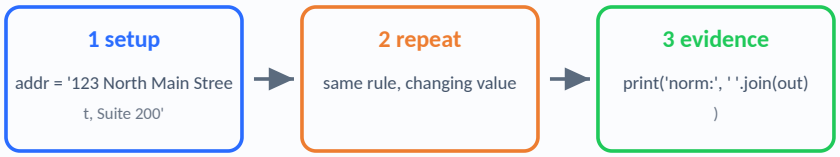
setup -> rule -> result

Logic Lens

- Setup: `addr = '123 North Main Street, Suite 200'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('norm:', ' '.join(out))` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`addr = '123 North Main Street, Suite 200'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`out = [repl.get(w, w) for w in parts]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('norm:', ' '.join(out))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `out = [repl.get(w, w) for w in parts]`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Normalize a Street Address: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: addr, North, Main, Street, Suite. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: addr = '123 North Main Street, Suite 200' | Core: out = [repl.get(w, w) for w in parts] | Evidence: print('norm:', ' '.join(out))

🛠️ Reader to Writer Bridge

1. Read: explain normalize a street address in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 404.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 406 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('norm:', ' '.join(out))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `addr = '123 North Main Street, Suite 200'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of normalize a street address and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Lowercasing, flattening commas to spaces, and substituting each token through an abbreviation dictionary turns free-form address text into a normalized token stream.

```
addr = '123 North Main Street, Suite 200'
repl = {'street': 'st', 'avenue': 'ave', 'north': 'n', 'suite': 'ste'}
parts = addr.lower().replace(',', ' ').split()
out = [repl.get(w, w) for w in parts]
print('raw: ', addr)
print('norm:', ' '.join(out))
```

```
output
raw:  123 North Main Street, Suite 200
norm: 123 n main st ste 200
```

How it works:

1. repl maps long forms like street and avenue to their short canonical abbreviations.
2. addr.lower().replace(',', ' ').split() lowercases, swaps the comma for a space, and breaks the address into a word list.
3. The comprehension calls repl.get(w, w) on each word, substituting the abbreviation when present and otherwise leaving the word alone.
4. Joining the rewritten tokens prints 123 n main st ste 200 beneath the raw address.

Try it:

1. Extend the repl map with road, boulevard, drive, and their abbreviations and test a new address.
2. Make the mapping case-insensitive while preserving the numeric house number untouched.
3. Add directional suffixes so southwest and northeast also fold to short forms.
4. Add 'road':'rd' and 'boulevard':'blvd' to the replacement map.

⚠ Common Mistakes

- Replacing on the raw string (not per-token) would turn 'streetlight' into 'stlight'.
- Forgetting to drop the comma leaves 'street,' which never matches the 'street' key.
- A one-way map ('street'->'st') will not normalize an address already written as 'st'.

Skills gained address normalization | token mapping | dictionary lookup

→ Where You'll Use This: Address normalization underpins NAP audits and duplicate detection, where North Main Street and N Main St must register as the same address rather than two competing listings.

⚡ Mental Model

Like a translator who rewrites every dialect of an address into one official shorthand the post office recognizes.

Project 406 Compare NAP Across Citations and Flag Mismatches

💡 The Idea Behind It

NAP means Name, Address, Phone, the trio that must agree everywhere you appear. This code holds a canonical record and a dictionary of citations, then for each source builds a sorted list of fields whose value differs from the canonical. An empty list prints match; otherwise it names exactly which fields drifted, so Yelp passes while YellowPages and Facebook each fail one field.

What You'll Build & Learn

Diff each external citation against a canonical NAP record and report precisely which fields are inconsistent per source.

- Comparing nested dictionaries field by field
- Building a list with a filtered comprehension
- Conditional expression inside a print

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 405: normalize a street address. Project 407 changes the surface problem while preserving the underlying method.

progress check

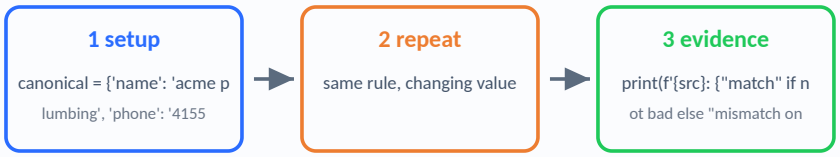
Reading target: compare nap across citations and flag mismatches is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `canonical = {'name': 'acme plumbing', 'phone': '4155550199', 'zip': '94110'}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print(f'{src}: {"match" if not bad else "mismatch on " + ",".join(bad)}')` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`canonical = {'name': 'acme plumbing', 'phone': '4155550199', 'zip': '94110'}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`bad = sorted(f for f in canonical if c[f] != canonical[f])`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{src}: {"match" if not bad else "mismatch on " + ",".join(bad)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `bad = sorted(f for f in canonical if c[f] != canonical[f])`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compare NAP Across Citations and Flag Mismatches: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: canonical, name, acme, plumbing, phone. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: canonical = {'name': 'acme plumbing', 'phone': '4155550199', 'zip': '94... | Core: bad = sorted(f for f in canonical if c[f] != canonical[f]) | Evidence: print(f'{src}: {"match" if not bad else "mismatch on " + ",".join(bad)}...

🛠️ Reader to Writer Bridge

1. Read: explain compare nap across citations and flag mismatches in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 405.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 407 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{src}: {"match" if not bad else "mismatch on " + ",".join(bad)}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `canonical = {'name': 'acme plumbing', 'phone': '4155550199', 'zip': '94110'}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compare nap across citations and flag mismatches and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Comparing each citation field to the canonical dict and collecting the keys that differ pinpoints the precise NAP fields that need fixing per source.

```
canonical = {'name': 'acme plumbing', 'phone': '4155550199', 'zip': '94110'}
citations = {
    'Yelp': {'name': 'acme plumbing', 'phone': '4155550199', 'zip': '94110'},
    'YellowPages': {'name': 'acme plumbing co', 'phone': '4155550199', 'zip':
'94110'},
    'Facebook': {'name': 'acme plumbing', 'phone': '4155550100', 'zip':
'94110'},
}
for src, c in citations.items():
    bad = sorted(f for f in canonical if c[f] != canonical[f])
    print(f'{src}: {"match" if not bad else "mismatch on " + ",".join(bad)}')
```

```
output
Yelp: match
YellowPages: mismatch on name
```

```
Facebook: mismatch on phone
```

How it works:

1. canonical defines the source-of-truth name, phone, and zip the listings should match.
2. citations maps three sources to their own NAP dictionaries, with deliberate drift in name and phone.
3. For each source, the comprehension collects sorted field names where c[f] differs from canonical[f].
4. The print emits match when bad is empty, otherwise mismatch on followed by the joined offending field names.

Try it:

1. Add an address field to the canonical and citation records and include it in the diff.
2. Normalize each value with the earlier helpers before comparing so formatting differences are forgiven.
3. Aggregate a count of how many sources mismatched on each individual field.
4. Add an 'address' field to every record and a citation that gets it wrong.

⚠ Common Mistakes

- Printing only 'mismatch' without the field names hides what actually needs fixing.
- Comparing whole dicts with `==` tells you there is a problem but not which field.
- Not sorting the bad-field list makes the message order nondeterministic across runs.

Skills gained NAP diffing | field comparison | mismatch reporting

→ Where You'll Use This: Citation-audit dashboards run exactly this diff across dozens of directories, telling an SEO which listings need editing to bring a clients name, address, and phone back into agreement.

Mental Model

Like proofreading photocopies against a master document, circling on each copy the lines that no longer match the original.

Project 407 Score Overall NAP Consistency

The Idea Behind It

Per-source diffs are useful, but managers want a single headline figure. Here the canonical NAP is a tuple, and each citation is a tuple compared with a plain equality test. Counting how many citations exactly equal the canonical, dividing by the total, and multiplying by one hundred yields a consistency percentage that summarizes overall listing health in one number.

What You'll Build & Learn

Compute the share of citations that exactly match the canonical NAP, producing a single consistency percentage.

- Comparing whole tuples for equality
- Counting matches with sum over a generator
- Converting a ratio to a rounded percentage

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 406: compare nap across citations and flag mismatches. Project 408 changes the surface problem while preserving the underlying method. *progress check*

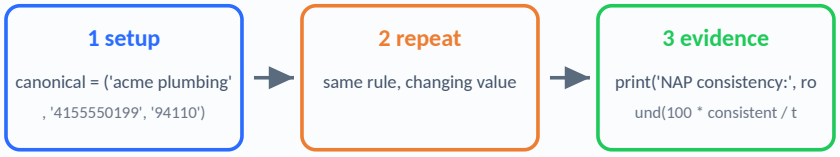
Reading target: score overall nap consistency is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `canonical = ('acme plumbing', '4155550199', '94110')` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('NAP consistency:', round(100 * consistent / total, 1), '%')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`canonical = ('acme plumbing', '4155550199', '94110')`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`consistent = sum(1 for c in citations.values() if c == canonical)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('NAP consistency:', round(100 * consistent / total, 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `consistent = sum(1 for c in citations.values() if c == canonical)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score Overall NAP Consistency: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: canonical, acme, plumbing, citations, Yelp. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: canonical = ('acme plumbing', '4155550199', '94110') | Core: consistent = sum(1 for c in citations.values() if c == canonical) | Evidence: print('NAP consistency:', round(100 * consistent / total, 1), '%')

Reader to Writer Bridge

1. Read: explain score overall nap consistency in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 406.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 408 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print('NAP consistency:', round(100 * consistent / total, 1), '%')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `canonical = ('acme plumbing', '4155550199', '94110')` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score overall nap consistency and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Counting citations whose full NAP tuple equals the canonical tuple, then dividing by the total, yields a single consistency percentage you can track over time.

```
canonical = ('acme plumbing', '4155550199', '94110')
citations = {
    'Yelp': ('acme plumbing', '4155550199', '94110'),
    'Bing': ('acme plumbing', '4155550199', '94110'),
    'Apple': ('acme plumbing co', '4155550199', '94110'),
    'Facebook': ('acme plumbing', '4155550100', '94110'),
    'Foursquare': ('acme plumbing', '4155550199', '94110'),
}
consistent = sum(1 for c in citations.values() if c == canonical)
total = len(citations)
print(f'consistent: {consistent}/{total}')
```

```
print(f'inconsistent: {total - consistent}/{total}')
print('NAP consistency:', round(100 * consistent / total, 1), '%')
```

```
output
consistent: 3/5
inconsistent: 2/5
NAP consistency: 60.0 %
```

How it works:

1. canonical is a three-element tuple of name, phone, and zip used as the reference.
2. citations maps five sources to tuples, two of which differ in name or phone.
3. consistent sums one for every citation tuple equal to canonical, giving three of five.
4. The final prints show consistent and inconsistent counts plus 100 times consistent over total rounded to 60.0 percent.

Try it:

1. Report which specific sources were inconsistent alongside the headline percentage.
2. Award partial credit so a citation matching two of three fields counts as two-thirds.
3. Raise an alert when consistency falls below a configurable threshold such as 80 percent.
4. Add a citation that matches and watch the percentage climb.

⚠ Common Mistakes

- Comparing tuples element-by-element by hand is error-prone; a direct `==` on tuples is exact.
- Dividing integers without `* 100` first can lose precision before the percentage is formed.
- Counting inconsistent as `total - consistent` is right only if every citation is classified once.

Skills gained consistency scoring | tuple equality | percentage math

→ Where You'll Use This: A NAP-consistency percentage is a core local-SEO key metric, tracked over time so an agency can show citation cleanup moving the number from 60 toward 100 percent.

⚡ Mental Model

Like a hygiene grade in a restaurant window: one percentage that distills many inspections into a figure anyone can read at a glance.

Project 408 Score Business-Profile Completeness

💡 The Idea Behind It

A profile stuffed with photos, hours, and a description outranks a skeleton one. This code defines the fields that matter, then splits them into present and missing using truthiness, so an empty description string counts as

missing while a list of categories counts as present. Dividing present by total gives a completeness percentage, here 85.7 with description the only gap.

What You'll Build & Learn

Score how complete a business profile is by counting which key fields hold a truthy value.

- Truthiness of strings, lists, and numbers
- Splitting fields into present and missing
- profile.get to read fields safely

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 407: score overall nap consistency. Project 409 changes the surface problem while preserving the underlying method.

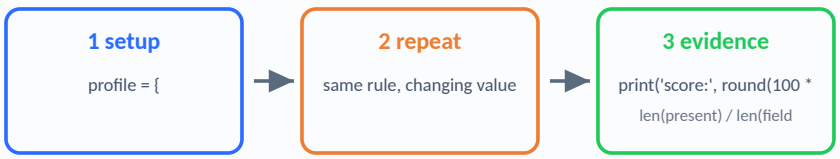
progress check

Reading target: score business-profile completeness is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `profile = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('score:', round(100 * len(present) / len(fields), 1), '%')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`profile = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`present = [f for f in fields if profile.get(f)]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('score:', round(100 * len(present) / len(fields), 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `present = [f for f in fields if profile.get(f)]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score Business-Profile Completeness: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: profile, name, Acme, Plumbing, hours. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: profile = { | Core: present = [f for f in fields if profile.get(f)] | Evidence: print('score:', round(100 * len(present) / len(fields), 1), '%')

🛠️ Reader to Writer Bridge

1. Read: explain score business-profile completeness in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 407.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 409 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('score:', round(100 * len(present) / len(fields), 1), '%')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `profile = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score business-profile completeness and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Treating any falsy field (empty string, zero, empty list) as absent and counting the truthy ones over the total field list produces a completeness score.

```
profile = {
    'name': 'Acme Plumbing',
    'hours': {'mon': '9–5'},
    'categories': ['Plumber'],
    'photos': 6,
    'description': '',
    'website': 'https://acme.example',
    'phone': '4155550199',
}
fields = ['name', 'hours', 'categories', 'photos', 'description', 'website',
'phone']
present = [f for f in fields if profile.get(f)]
missing = [f for f in fields if not profile.get(f)]
print('present:', len(present), 'of', len(fields))
print('missing:', missing)
print('score:', round(100 * len(present) / len(fields), 1), '%')
```

```
output
```

```
present: 6 of 7
missing: ['description']
score: 85.7 %
```

How it works:

1. profile holds seven fields, with description set to an empty string that reads as falsy.
2. fields lists the seven keys the score considers.
3. present keeps fields where profile.get(f) is truthy; missing keeps the rest, capturing only description.
4. The prints show six of seven present, the missing list, and 100 times six over seven rounded to 85.7 percent.

Try it:

1. Weight some fields heavier so a missing phone hurts the score more than a missing description.
2. Treat a photos count below a minimum threshold as missing rather than only zero.
3. Print a prioritized to-do list of the missing fields ordered by importance.
4. Fill in the description and watch the score reach 100%.

⚠ Common Mistakes

- Checking `f in profile` instead of `profile.get(f)` counts an empty-string field as present.
- Photos as `0` is falsy and correctly counts as missing; a non-empty list of zero photos would not.
- Hard-coding the denominator instead of `len(fields)` drifts when you add a field.

Skills gained completeness scoring | truthiness filtering | profile auditing

→ Where You'll Use This: Completeness scores drive client recommendations: an agency points to the missing description or empty hours as concrete, fillable gaps that lift a profiles standing in local results.

⚡ Mental Model

Like a checklist taped to a job site where every blank box is a task standing between you and sign-off.

Project 409 Check Category Relevance

💡 The Idea Behind It

Google penalizes business profiles tagged with off-topic categories. The code defines a primary category and a chosen list, then checks each chosen category for membership in an allow-list of relevant ones. Categories outside the set are labeled OFF-TOPIC, the primary is starred, and the off-topic count surfaces stray picks like Coffee Shop sitting on a plumbers profile.

What You'll Build & Learn

Flag categories that fall outside an allow-list of relevant ones and count the off-topic picks on a profile.

- Set membership testing with in
- Tagging items by a condition
- Counting violations with a comprehension

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 408: score business-profile completeness. Project 410 changes the surface problem while preserving the underlying method. *progress check*

Reading target: check category relevance is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `primary = 'Plumber'` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('off-topic count:', len(off))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`primary = 'Plumber'`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.

Move	Code to inspect	Why it matters
2. Apply the rule	`tag = 'relevant' if cat in relevant else 'OFF-TOPIC'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('off-topic count:', len(off))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `tag = 'relevant' if cat in relevant else 'OFF-TOPIC'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Category Relevance: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: primary, Plumber, chosen, Drain, Cleaning. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: primary = 'Plumber' | Core: tag = 'relevant' if cat in relevant else 'OFF-TOPIC' | Evidence: print('off-topic count:', len(off))

Reader to Writer Bridge

1. Read: explain check category relevance in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 408.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 410 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('off-topic count:', len(off))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `primary = 'Plumber'` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check category relevance and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Membership tests against a relevant-category set classify each chosen category, and flagging the primary separately mirrors how the primary category carries the most ranking weight.

```
primary = 'Plumber'
chosen = ['Plumber', 'Drain Cleaning Service', 'Coffee Shop', 'Contractor']
relevant = {'Plumber', 'Drain Cleaning Service', 'Contractor', 'Water Heater
Installation'}
for cat in chosen:
    tag = 'relevant' if cat in relevant else 'OFF-TOPIC'
    star = ' (primary)' if cat == primary else ''
    print(f'{cat}: {tag}{star}')
off = [c for c in chosen if c not in relevant]
print('off-topic count:', len(off))
```

```
output
Plumber: relevant (primary)
Drain Cleaning Service: relevant
Coffee Shop: OFF-TOPIC
Contractor: relevant
off-topic count: 1
```

How it works:

1. primary names the main category, and chosen lists four categories assigned to the profile.
2. relevant is a set of categories considered on-topic for this business.
3. The loop tags each chosen category relevant or OFF-TOPIC by testing cat in relevant, adding a primary star when it matches primary.
4. off collects chosen categories not in relevant, and its length prints an off-topic count of one for Coffee Shop.

Try it:

1. Verify the primary category itself is inside the relevant set and warn if it is not.
2. Suggest relevant categories the profile is missing by diffing chosen against relevant.
3. Load the relevant set from an external list keyed by industry.
4. Add 'Emergency Plumber' to the relevant set and a chosen category to match it.

⚠ Common Mistakes

- Using a list instead of a set for `relevant` makes the membership check O(n) and slower.
- Marking the primary by position instead of by value breaks if the order changes.
- Case-sensitive matching misses 'plumber' versus 'Plumber' unless both sides are normalized.

Skills gained category auditing | allow-list check | relevance scoring

→ Where You'll Use This: Category audits keep profiles compliant: SEOs compare a clients selected categories against a vetted relevant set to remove mismatches that risk a suspension or muddy ranking signals.

Mental Model

Like a bouncer with a guest list: every category on the profile is checked against who belongs, and gatecrashers get flagged.

Project 410 Compute Review Count and Average Rating

The Idea Behind It

The two loudest review signals are how many reviews you have and their average rating. This code takes a list of star ratings, counts them with len, averages with sum over n, and rounds to two places, then rounds again to a whole star. Those numbers feed both the local pack and the star snippet shown in search results.

What You'll Build & Learn

Compute the review count, the average star rating, and a rounded whole-star value from a list of ratings.

- Averaging with sum divided by len
- Rounding to a chosen precision
- Distinguishing count from average

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 409: check category relevance. Project 411 changes the surface problem while preserving the underlying method.

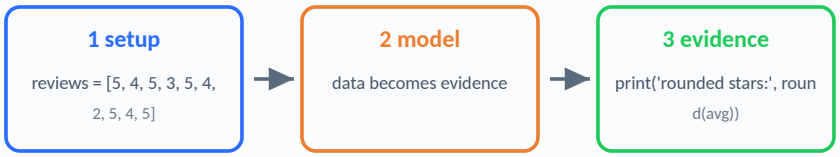
progress check

Reading target: compute review count and average rating is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `reviews = [5, 4, 5, 3, 5, 4, 2, 5, 4, 5]` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as data -> transformation -> evidence, not as isolated library calls.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('rounded stars:', round(avg))` exposes the result; match it against the output before changing the program. output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, model applies the chapter idea, and evidence proves your trace against the output. visual trace

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`reviews = [5, 4, 5, 3, 5, 4, 2, 5, 4, 5]`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`print('average rating:', round(avg, 2))`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. slow line
3. Check the evidence	`print('rounded stars:', round(avg))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('average rating:', round(avg, 2))`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Compute Review Count and Average Rating: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: reviews, len, avg, sum, review. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: reviews = [5, 4, 5, 3, 5, 4, 2, 5, 4, 5] | Core: print('average rating:', round(avg, 2)) | Evidence: print('rounded stars:', round(avg))

🛠️ Reader to Writer Bridge

1. Read: explain compute review count and average rating in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 409.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 411 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('rounded stars:', round(avg))`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `reviews = [5, 4, 5, 3, 5, 4, 2, 5, 4, 5]` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute review count and average rating and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The count is the list length and the average is the sum over that length; rounding to two places keeps the printed float stable.

```
reviews = [5, 4, 5, 3, 5, 4, 2, 5, 4, 5]
n = len(reviews)
avg = sum(reviews) / n
print('review count:', n)
print('average rating:', round(avg, 2))
print('rounded stars:', round(avg))
```

```
output
review count: 10
average rating: 4.2
rounded stars: 4
```

How it works:

1. reviews is a list of ten integer star ratings.
2. n = len(reviews) captures the review count.
3. avg = sum(reviews) / n computes the mean, which round(avg, 2) trims to 4.2.
4. round(avg) produces the whole-star value of 4 shown as rounded stars.

Try it:

1. Compute the median rating alongside the mean to resist outliers.
2. Weight recent reviews more heavily than old ones in the average.
3. Print the average to one decimal place using an f-string format spec.
4. Add a one-star review and watch the average fall.

⚠ Common Mistakes

- Dividing by a hard-coded count instead of `len(reviews)` breaks when reviews change.
- Printing the raw average gives a long float like 4.2000000001 on some inputs; round it.
- Confusing the rounded star (4) with the precise average (4.2) when displaying.

Skills gained rating average review counting rounding

→ Where You'll Use This: Review count and average rating are the headline figures in any local-SEO scorecard, mirroring the star and review-count badge that Google shows beside a listing in the map pack.

Mental Model

Like reading a thermometer and a crowd headcount at once: one tells you how warm the room feels, the other how full it is.

Project 411 Build a Rating Distribution Histogram

The Idea Behind It

An average hides the shape of opinion, since a 4.2 from all fours and fives differs from a 4.2 spiked with ones. This code uses Counter to tally ratings, then loops stars from five down to one, printing each count and a hash-mark bar of that length. The text histogram exposes the real distribution behind the mean.

What You'll Build & Learn

Build a text histogram of review ratings so the spread of stars is visible, not just the average.

- Tallying values with collections.Counter
- Iterating a descending range
- Drawing a bar by repeating a character

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 410: compute review count and average rating. Project 412 changes the surface problem while preserving the underlying method. *progress check*

Reading target: build a rating distribution histogram is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.

- Evidence: `print('total:', n)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`for star in range(5, 0, -1):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('total:', n)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for star in range(5, 0, -1):`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a Rating Distribution Histogram: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: collections, Counter, reviews, dist, len. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: for star in range(5, 0, -1): | Evidence: print('total:', n)

🛠️ Reader to Writer Bridge

1. Read: explain build a rating distribution histogram in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 410.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 412 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('total:', n)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a rating distribution histogram and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A `Counter` tallies each star value, and iterating from 5 down to 1 prints a fixed-order bar chart that is deterministic regardless of input order.

```
from collections import Counter
reviews = [5, 4, 5, 3, 5, 4, 2, 5, 4, 5, 1, 5]
dist = Counter(reviews)
n = len(reviews)
for star in range(5, 0, -1):
```

```
    c = dist.get(star, 0)
    bar = '#' * c
    print(f'{star}* {c:2} {bar}')
print('total:', n)
```

```
output
5*  6 ######
4*  3 ###
3*  1 #
2*  1 #
1*  1 #
total: 12
```

How it works:

1. Counter(reviews) tallies how many times each star value appears in the twelve-rating list.
2. The loop runs star from 5 down to 1 with range(5, 0, -1).
3. dist.get(star, 0) fetches the count for each star, defaulting to zero when absent.
4. '#' * c builds a bar whose width equals the count, and the f-string aligns the number before printing the total.

Try it:

1. Show the percentage of reviews each star represents next to its bar.
2. Scale the bar so the longest count fits a fixed width regardless of volume.
3. Highlight any star whose count exceeds a concerning threshold.
4. Print the percentage each star represents next to its bar.

⚠ Common Mistakes

- Iterating `dist` directly prints stars in insertion order, not 5-to-1.
- Using `dist[star]` instead of `dist.get(star, 0)` raises nothing but a missing star would be absent if you looped over keys.
- Building bars by string multiplication with a float count would raise a TypeError.

Skills gained rating histogram | Counter tallying | ascii bar chart

→ Where You'll Use This: Distribution histograms help diagnose review health: a fat one-star bar behind a decent average warns an SEO that hidden dissatisfaction could erode trust and click-through over time.

⚡ Mental Model

Like sorting a jar of coins into stacks by denomination so you see at a glance whether it is mostly quarters or pennies.

Project 412 Measure Review Velocity Over Months

The Idea Behind It

A steady drip of fresh reviews signals an active, trusted business. This code stores monthly review counts in a dictionary, sums and averages them, then splits the six months into a first quarter and last quarter, averaging each half. Comparing the two halves labels the trend accelerating or slowing, here climbing from five to ten per month.

What You'll Build & Learn

Measure whether review velocity is accelerating or slowing by comparing the first half of months to the last.

- Reading values from a dict in order
- Slicing a list into halves
- Comparing period averages for a trend

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 411: build a rating distribution histogram. Project 413 changes the surface problem while preserving the underlying method.

progress check

Reading target: measure review velocity over months is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

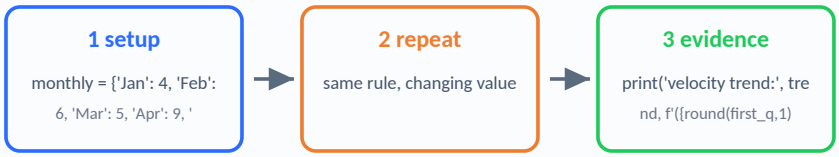
- Setup: `monthly = {'Jan': 4, 'Feb': 6, 'Mar': 5, 'Apr': 9, 'May': 11, 'Jun': 10}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('velocity trend:', trend, f'({round(first_q,1)} -> {round(last_q,1)})')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`monthly = {'Jan': 4, 'Feb': 6, 'Mar': 5, 'Apr': 9, 'May': 11, 'Jun': 10}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`trend = 'accelerating' if last_q > first_q else 'slowing'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('velocity trend:', trend, f'({round(first_q,1)} -> {round(last_q,1)})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `trend = 'accelerating' if last_q > first_q else 'slowing'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Arrow/scope tokens: Often connect a parameter to a body, a type to a return value, or a namespace to a member.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure Review Velocity Over Months: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: monthly, Jan, Feb, Mar, Apr. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: monthly = {'Jan': 4, 'Feb': 6, 'Mar': 5, 'Apr': 9, 'May': 11, 'Jun': 10} | Core: trend = 'accelerating' if last_q > first_q else 'slowing' | Evidence: print('velocity trend:', trend, f'({round(first_q,1)} -> {round(last_q,...

🛠️ Reader to Writer Bridge

1. Read: explain measure review velocity over months in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 411.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 413 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('velocity trend:', trend, f'({round(first_q,1)} -> {round(last_q,1)})')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `monthly = {'Jan': 4, 'Feb': 6, 'Mar': 5, 'Apr': 9, 'May': 11, 'Jun': 10}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure review velocity over months and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Splitting the monthly counts into two halves and comparing their averages turns raw counts into a simple accelerating-versus-slowing trend.

```
monthly = {'Jan': 4, 'Feb': 6, 'Mar': 5, 'Apr': 9, 'May': 11, 'Jun': 10}
months = list(monthly)
counts = [monthly[m] for m in months]
```

```
total = sum(counts)
avg = total / len(months)
first_q = sum(counts[:3]) / 3
last_q = sum(counts[3:]) / 3
trend = 'accelerating' if last_q > first_q else 'slowing'
print('total reviews:', total)
print('avg per month:', round(avg, 2))
print('velocity trend:', trend, f'({round(first_q,1)} -> {round(last_q,1)})')
```

```
output
total reviews: 45
avg per month: 7.5
velocity trend: accelerating (5.0 -> 10.0)
```

How it works:

1. monthly maps six months to their review counts, and counts pulls those values in order.
2. total and avg give the overall sum and mean per month.
3. first_q averages counts[:3] and last_q averages counts[3:], the two halves of the period.
4. trend reads accelerating when last_q exceeds first_q, and the print shows the 5.0 to 10.0 climb.

Try it:

1. Compute a month-over-month percentage change instead of comparing two halves.
2. Flag any single month whose count drops sharply versus the prior one.
3. Extend to twelve months and compare quarter-by-quarter instead of halves.
4. Add three more months and adjust the quarter split.

⚠ Common Mistakes

- Hard-coding the split index instead of deriving it makes the halves wrong if months change.
- Comparing totals rather than averages would mislead if the halves had different lengths.
- Reading `monthly` order matters: dicts preserve insertion order here, but sorting keys would scramble the timeline.

Skills gained review velocity | trend detection | list slicing

→ Where You'll Use This: Velocity tracking tells an SEO whether a review-generation campaign is working, since a rising fresh-review rate is a freshness signal that supports local-pack ranking.

⚡ Mental Model

Like checking a faucet by watching whether the drips are quickening or trailing off rather than counting the total puddle.

Project 413 Compute Review Respond Rate

The Idea Behind It

Replying to reviews, especially angry ones, is both a trust and a ranking signal. This code holds reviews with a rating and a replied flag, counts overall replies over the total, then isolates reviews rated two or below and computes their reply rate separately. The negative respond rate matters more, and here it reaches a full hundred percent.

What You'll Build & Learn

Compute the overall review respond rate and the more important negative-review respond rate from a list of reviews.

- Counting boolean flags with sum
- Filtering reviews by a rating threshold
- Computing two rates from one list

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 412: measure review velocity over months. Project 414 changes the surface problem while preserving the underlying method.

progress check

Reading target: compute review respond rate is the surface task; the transferable Python move is to mark setup, rule, and output before running.

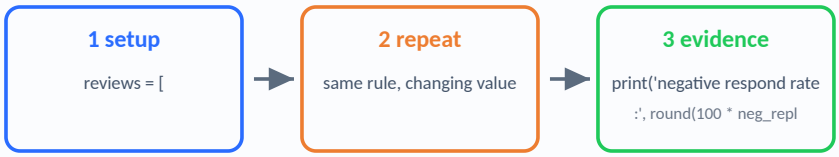
setup -> rule -> result

Logic Lens

- Setup: `reviews = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('negative respond rate:', round(100 * neg_replied / len(neg), 1), '%')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`reviews = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`replied = sum(1 for r in reviews if r['replied'])`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('negative respond rate:', round(100 * neg_replied / len(neg), 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `replied = sum(1 for r in reviews if r['replied'])`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute Review Respond Rate: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: reviews, rating, replied, sum, len. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: reviews = [| Core: replied = sum(1 for r in reviews if r['replied']) | Evidence: print('negative respond rate:', round(100 * neg_replied / len(neg), 1),...

🛠️ Reader to Writer Bridge

1. Read: explain compute review respond rate in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 412.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 414 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('negative respond rate:', round(100 * neg_replied / len(neg), 1), '%')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `reviews = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute review respond rate and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Counting replied reviews over the total gives the overall rate, while filtering to ratings of two or below isolates the negative respond rate that matters most.

```
reviews = [
    {'rating': 5, 'replied': True},
    {'rating': 1, 'replied': True},
    {'rating': 4, 'replied': False},
    {'rating': 2, 'replied': True},
    {'rating': 5, 'replied': False},
```

```
]
replied = sum(1 for r in reviews if r['replied'])
n = len(reviews)
neg = [r for r in reviews if r['rating'] <= 2]
neg_replied = sum(1 for r in neg if r['replied'])
print('overall respond rate:', round(100 * replied / n, 1), '%')
print('negative respond rate:', round(100 * neg_replied / len(neg), 1), '%')
```

```
output
overall respond rate: 60.0 %
negative respond rate: 100.0 %
```

How it works:

1. reviews is a list of dicts each carrying a rating and a replied boolean.
2. replied sums the True flags, and dividing by n yields the overall respond rate of 60 percent.
3. neg filters reviews where rating is two or below, capturing the two negatives.
4. neg_replied counts replies among those, and dividing prints the negative respond rate of 100 percent.

Try it:

1. Add a response-time field and report the average days to reply.
2. Break the respond rate out by star level rather than only negative versus all.
3. Guard against a division error when there are no negative reviews at all.
4. Add an unreplied one-star review and watch the negative rate drop.

⚠ Common Mistakes

- Dividing by `len(reviews)` for the negative rate instead of `len(neg)` understates it badly.
- Treating a missing 'replied' key as False requires `.get`; here every review has it.
- An empty negative list would divide by zero; guard it when data can lack negatives.

Skills gained respond rate | boolean counting | negative filtering

→ Where You'll Use This: Respond-rate metrics shape reputation-management work, where promptly answering one and two-star reviews is the lever an SEO pulls to rebuild trust and signal an engaged owner to Google.

⚡ Mental Model

Like grading a shop on how often it answers the phone, with extra weight on whether it picks up when a customer is upset.

Project 414 Rank Listings by Proximity to the Searcher

💡 The Idea Behind It

Local results lean hard on distance, so you need kilometres from coordinates without a mapping library. This code approximates them with flat-earth scaling: latitude and longitude differences are multiplied by 111 km, with the longitude term shrunk by the cosine of the latitude. Pythagoras combines the legs, and sorting by that distance ranks businesses by nearness to the searcher.

What You'll Build & Learn

Approximate distance in kilometres from latitude and longitude and rank businesses by how near they are.

- Flat-earth degree-to-km scaling
- Cosine correction for longitude
- Sorting with a distance key function

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 413: compute review respond rate. Project 415 changes the surface problem while preserving the underlying method.

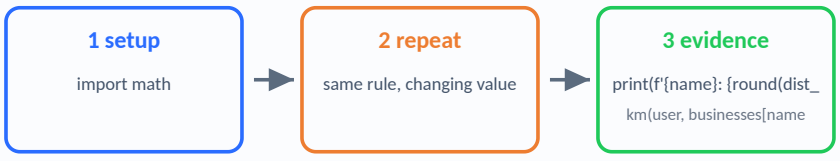
progress check

Reading target: rank listings by proximity to the searcher is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import math` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print(f'{name}: {round(dist_km(user, businesses[name]), 2)} km')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import math`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`for name in sorted(businesses, key=lambda n: dist_km(user, businesses[n])):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{name}: {round(dist_km(user, businesses[name]), 2)} km')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for name in sorted(businesses, key=lambda n: dist_km(user, businesses[n])):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Rank Listings by Proximity to the Searcher: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: math, user, businesses, dist_km, dlat. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.

Layer	What to read
Evidence	Entry: import math | Core: for name in sorted(businesses, key=lambda n: dist_km(user, businesses[n... | Evidence: print(f'{name}: {round(dist_km(user, businesses[name]), 2)} km')

🛠️ Reader to Writer Bridge

1. Read: explain rank listings by proximity to the searcher in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 413.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 415 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{name}: {round(dist_km(user, businesses[name]), 2)} km')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import math` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of rank listings by proximity to the searcher and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Scaling latitude and longitude deltas to kilometres and taking the Euclidean distance gives a fast proximity estimate, accurate enough for local ranking over short spans.

```
import math
user = (37.7749, -122.4194)
businesses = {
    'A': (37.7750, -122.4180),
    'B': (37.8044, -122.2712),
    'C': (37.7849, -122.4094),
}
def dist_km(a, b):
    dlat = (a[0] - b[0]) * 111.0
    dlon = (a[1] - b[1]) * 111.0 * math.cos(math.radians(a[0]))
    return math.sqrt(dlat * dlat + dlon * dlon)
```

```
    for name in sorted(businesses, key=lambda n: dist_km(user, businesses[n])):
        print(f'{name}: {round(dist_km(user, businesses[name]), 2)} km')
```

```
output
A: 0.12 km
C: 1.41 km
B: 13.41 km
```

How it works:

1. user holds the searchers coordinates, and businesses maps three names to their own coordinates.
2. dist_km scales the latitude gap by 111 km and the longitude gap by 111 times the cosine of the latitude.
3. math.sqrt of the squared legs gives the planar distance in kilometres.
4. sorted orders names by dist_km, printing A nearest at 0.12 km and B farthest at 13.41 km.

Try it:

1. Replace the flat-earth math with the full haversine formula and compare results.
2. Filter out businesses beyond a service radius before ranking the rest.
3. Return distances in miles by adjusting the per-degree constant.
4. Move the user's point and watch the ranking reshuffle.

⚠ Common Mistakes

- Forgetting the `cos(lat)` longitude correction overstates east-west distance away from the equator.
- Using raw degree differences as if they were kilometres mixes units entirely.
- Sorting by `dist_km` recomputes it per comparison; caching distances first is cheaper on big lists.

Skills gained proximity ranking | geo distance math | key-based sorting

→ Where You'll Use This: Distance estimation explains the proximity factor in local ranking, letting an SEO model why a nearer competitor outranks a stronger but farther one for a given searchers location.

⚡ Mental Model

Like measuring a city map with a ruler and a fudge factor rather than walking every street with an odometer.

Project 415 Check Local-Pack Eligibility

💡 The Idea Behind It

To surface in the map pack a profile generally needs a rating, enough reviews, a category, and verification. This code packs those four conditions into a checks dictionary of booleans, prints each as yes or no, then folds them with all so eligibility is True only when every gate passes. It reproduces the eligibility screen in a few lines.

What You'll Build & Learn

Reproduce a local-pack eligibility check by requiring a rating, enough reviews, a category, and verification.

- Building a dict of boolean checks
- Combining conditions with all
- Threshold comparison on a count

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 414: rank listings by proximity to the searcher. Project 416 changes the surface problem while preserving the underlying method.

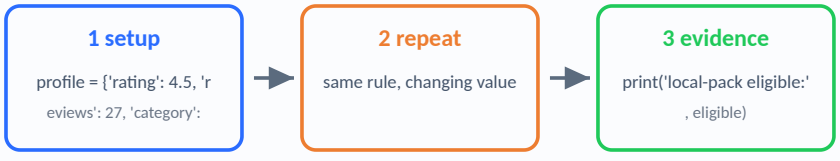
progress check

Reading target: check local-pack eligibility is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `profile = {'rating': 4.5, 'reviews': 27, 'category': 'Plumber', 'verified': True}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('local-pack eligible:', eligible)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`profile = {'rating': 4.5, 'reviews': 27, 'category': 'Plumber', 'verified': True}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`print(f'{label}: {"yes" if ok else "no"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('local-pack eligible:', eligible)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{label}: {"yes" if ok else "no"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check Local-Pack Eligibility: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: profile, rating, reviews, category, Plumber. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: profile = {'rating': 4.5, 'reviews': 27, 'category': 'Plumber', 'verifi... | Core: print(f'{label}: {"yes" if ok else "no"}') | Evidence: print('local-pack eligible:', eligible)

Reader to Writer Bridge

1. Read: explain check local-pack eligibility in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 414.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 416 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('local-pack eligible:', eligible)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `profile = {'rating': 4.5, 'reviews': 27, 'category': 'Plumber', 'verified': True}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of check local-pack eligibility and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Building a dict of named boolean checks makes the verdict transparent (you can print each one) and `all(...)` collapses them into a single eligibility decision.

```
profile = {'rating': 4.5, 'reviews': 27, 'category': 'Plumber', 'verified':
True}
checks = {
    'has rating': profile['rating'] > 0,
    'has reviews': profile['reviews'] >= 5,
    'has category': bool(profile['category']),
    'verified': profile['verified'],
}
for label, ok in checks.items():
    print(f'{label}: {"yes" if ok else "no"}')
eligible = all(checks.values())
print('local-pack eligible:', eligible)
```

```
output
has rating: yes
has reviews: yes
has category: yes
verified: yes
local-pack eligible: True
```

How it works:

1. profile holds a rating, review count, category, and verified flag.
2. checks maps four labels to boolean expressions, including reviews greater than or equal to five.
3. The loop prints each check as yes or no based on its boolean value.
4. all(checks.values()) is True only when every gate passes, so eligible prints True here.

Try it:

1. Add a checks entry requiring a complete address and re-evaluate eligibility.
2. Report which specific gates failed instead of only a single True or False.
3. Make the minimum review count configurable rather than hard-coded at five.
4. Drop the review count below the threshold and watch eligibility fail.

⚠ Common Mistakes

- Using `or` instead of `all` would pass a profile that meets just one condition.
- A reviews threshold of `> 0` is too lax; the pack favours a minimum like five.
- Treating `rating == 0` as valid lets an unrated profile slip through.

Skills gained eligibility gating | boolean aggregation | threshold check

→ Where You'll Use This: Eligibility gating mirrors the prerequisites Google effectively enforces for map-pack inclusion, helping an SEO confirm a new profile clears the basics before chasing finer ranking factors.

⚡ Mental Model

Like a ride-height bar at a theme park: you only get on when every requirement is cleared at once, not most of them.

Project 416 Detect Duplicate Listings

💡 The Idea Behind It

Duplicate profiles split reviews and confuse Google about which listing is real. This code groups listings by a key of their name and phone using setdefault to collect row indices, then keeps only keys appearing more than once. Because acme plumbing co normalizes differently, only the two identical acme plumbing rows surface as a duplicate pair.

What You'll Build & Learn

Surface duplicate listings by grouping rows under a name-and-phone key and reporting keys with more than one row.

- Grouping with dict setdefault

- Tracking row indices during a loop
- Filtering groups by size

🧭 Skill Roadmap

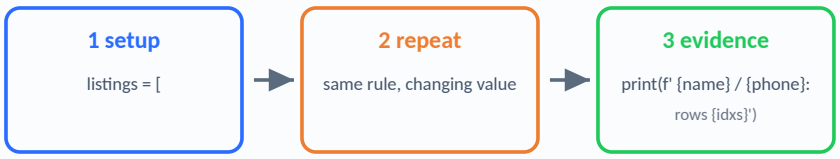
Skill step: moves you from recognition toward independent tracing. It builds on Project 415: check local-pack eligibility. Project 417 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect duplicate listings is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `listings = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print(f' {name} / {phone}: rows {idxs}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`listings = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`dups = {k: v for k, v in seen.items() if len(v) > 1}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f' {name} / {phone}: rows {idxs}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `dups = {k: v for k, v in seen.items() if len(v) > 1}`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Duplicate Listings: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: listings, name, acme, plumbing, phone. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: listings = [| Core: dups = {k: v for k, v in seen.items() if len(v) > 1} | Evidence: print(f' {name} / {phone}: rows {idxs}')

Reader to Writer Bridge

1. Read: explain detect duplicate listings in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 415.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 417 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f' {name} / {phone}: rows {idxs}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `listings = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect duplicate listings and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Using `setdefault` to bucket row indices under each (name, phone) key, then keeping buckets of length over one, reveals exactly which listings duplicate each other.

```
listings = [
    {'name': 'acme plumbing', 'phone': '4155550199'},
    {'name': 'acme plumbing', 'phone': '4155550199'},
    {'name': 'acme plumbing co', 'phone': '4155550199'},
    {'name': 'bobs cafe', 'phone': '4155550111'},
]
seen = {}
for i, l in enumerate(listings):
    key = (l['name'], l['phone'])
    seen.setdefault(key, []).append(i)
dups = {k: v for k, v in seen.items() if len(v) > 1}
print('duplicate listings:')
for (name, phone), idxs in sorted(dups.items()):
    print(f'  {name} / {phone}: rows {idxs}')
```

```
output
duplicate listings:
  acme plumbing / 4155550199: rows [0, 1]
```

How it works:

1. listings is a list of four dicts, two of which share the exact same name and phone.
2. The enumerate loop builds a key tuple of name and phone and appends each row index via seen.setdefault(key, []).
3. dups keeps only keys whose index list has more than one entry.
4. The sorted print shows acme plumbing at 4155550199 occupying rows 0 and 1.

Try it:

1. Normalize the name and phone with the earlier helpers before keying so near-duplicates also group.
2. Key on name and address instead of phone and compare which finds more duplicates.
3. Print a total count of duplicate groups and the rows involved.
4. Normalize the name first so the 'co' suffix row joins the duplicate group.

⚠ Common Mistakes

- Keying on the raw dict (unhashable) instead of a tuple raises a TypeError.
- Not normalizing 'acme plumbing co' first means a real duplicate with a suffix is missed.
- Reporting any repeated name alone would merge two genuinely different businesses sharing a name.

Skills gained duplicate detection | dict grouping | key building

→ Where You'll Use This: Duplicate detection is a staple of listing cleanup, where an SEO finds and merges redundant profiles so reviews and signals consolidate onto one canonical listing rather than scattering.

Mental Model

Like sorting mail into pigeonholes by recipient, then flagging any slot that ended up with two letters for the same person.

Project 417 Define and Audit a Service Area

The Idea Behind It

Service-area businesses declare the zip codes they cover instead of a storefront. This code takes a list with duplicate zips, dedupes it into a sorted set, reports the declared versus unique counts, and tests whether a target zip is a member. Membership confirms 94107 sits inside the declared area while exposing that two declared entries were repeats.

What You'll Build & Learn

Dedupe a service-area zip list and test whether a target zip falls inside the declared coverage.

- Deduplicating with set
- Sorting unique values
- Membership testing with in

Skill Roadmap

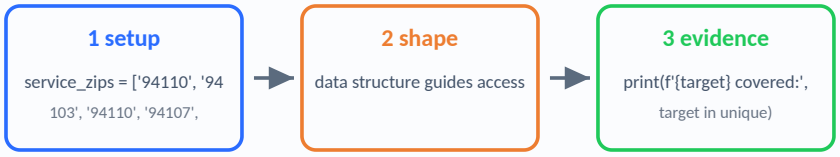
Skill step: moves you from recognition toward independent tracing. It builds on Project 416: detect duplicate listings. Project 418 changes the surface problem while preserving the underlying method. *progress check*

Reading target: define and audit a service area is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `service_zips = ['94110', '94103', '94110', '94107', '94103']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The data structure is doing part of the thinking. Ask what the collection stores, how items are found, and what order or uniqueness it promises.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print(f'{target} covered:', target in unique)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, shape applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`service_zips = ['94110', '94103', '94110', '94107', '94103']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`target = '94107'`	This line depends on the chosen data shape. Ask how items are stored, found, or ordered. *slow line*
3. Check the evidence	`print(f'{target} covered:', target in unique)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `target = '94107'`

Construction pattern

The construction is setup -> operation -> evidence. Name those three parts before worrying about every token.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Name the collection shape, then ask how an item is added, selected, transformed, or counted.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Define and Audit a Service Area: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: service_zips, target, unique, sorted, declared. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: service_zips = ['94110', '94103', '94110', '94107', '94103'] | Core: target = '94107' | Evidence: print(f'{target} covered:', target in unique)

🛠️ Reader to Writer Bridge

1. Read: explain define and audit a service area in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 416.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 418 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{target} covered:', target in unique)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `service_zips = ['94110', '94103', '94110', '94107', '94103']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of define and audit a service area and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A `set` removes accidental duplicate zips, `sorted` makes the printed list deterministic, and an `in` test answers coverage for any target zip.

```
service_zips = ['94110', '94103', '94110', '94107', '94103']
target = '94107'
unique = sorted(set(service_zips))
print('declared zips:', len(service_zips))
print('unique zips:', unique)
print('count:', len(unique))
print(f'{target} covered:', target in unique)
```

```
output
declared zips: 5
unique zips: ['94103', '94107', '94110']
count: 3
94107 covered: True
```

How it works:

1. service_zips lists five declared zips, with 94110 and 94103 each appearing twice.
2. unique = sorted(set(service_zips)) collapses duplicates and orders the result.
3. The prints show the declared count of five against the unique count of three.
4. target in unique tests membership, printing that 94107 is covered as True.

Try it:

1. Report which declared zips were duplicates rather than only the counts.
2. Test a list of target zips at once and print coverage for each.
3. Flag a service area that declares far more zips than is realistic.
4. Add a zip outside the area and test a target that is not covered.

⚠ Common Mistakes

- Counting `len(service_zips)` as the coverage size double-counts the repeated zips.
- Printing the set unsorted gives a nondeterministic order.
- Comparing zips as integers drops a leading zero like '07030'.

Skills gained service-area audit set dedup membership test

→ Where You'll Use This: Service-area auditing supports plumbers, electricians, and other no-storefront businesses, letting an SEO verify a clients declared zips actually cover the neighborhoods they want to rank in.

Mental Model

Like checking a pizza delivery map: you collapse the scribbled-over streets into a clean boundary and ask whether one address falls inside it.

Project 418 Parse and Validate Opening Hours

The Idea Behind It

Opening hours like 09:00-17:00 must parse into an open and a close time, and close must come after open. This code loops the hours dict, prints closed days outright, otherwise splits the range on the dash and tests whether the open string sorts before the close string. The inverted Friday range 18:00-09:00 is correctly flagged invalid.

What You'll Build & Learn

Parse each days opening hours into open and close times and flag any range where close precedes open.

- Splitting a string on a delimiter
- Skipping with continue on a sentinel
- Lexicographic time comparison

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 417: define and audit a service area. Project 419 changes the surface problem while preserving the underlying method.

progress check

Reading target: parse and validate opening hours is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

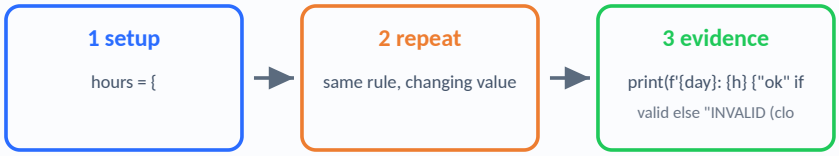
Logic Lens

- Setup: `hours = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.

- Evidence: `print(f'{day}: {h} {"ok" if valid else "INVALID (close before open)"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`hours = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`if h == 'closed':`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{day}: {h} {"ok" if valid else "INVALID (close before open)"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if h == 'closed':`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Parse and Validate Opening Hours: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: hours, mon, tue, wed, closed. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: hours = { | Core: if h == 'closed': | Evidence: print(f'{day}: {h} {"ok" if valid else "INVALID (close before open)"}')

🛠️ Reader to Writer Bridge

1. Read: explain parse and validate opening hours in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 417.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 419 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{day}: {h} {"ok" if valid else "INVALID (close before open)"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `hours = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of parse and validate opening hours and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Splitting each range on the dash and comparing the zero-padded time strings lexically (which equals chronological order for HH:MM) validates the interval without parsing to numbers.

```
hours = {
    'mon': '09:00-17:00',
    'tue': '09:00-17:00',
    'wed': 'closed',
    'thu': '09:00-12:00',
    'fri': '18:00-09:00',
}
for day, h in hours.items():
    if h == 'closed':
        print(f'{day}: closed')
        continue
    o, c = h.split('-')
    valid = o < c
    print(f'{day}: {h} {"ok" if valid else "INVALID (close before open)"}')
```

```
output
mon: 09:00-17:00 ok
tue: 09:00-17:00 ok
wed: closed
thu: 09:00-12:00 ok
fri: 18:00-09:00 INVALID (close before open)
```

How it works:

1. hours maps weekdays to time ranges, with wed marked closed and fri inverted.
2. When h equals closed the code prints closed and continues to the next day.
3. o, c = h.split('-') breaks the range into open and close strings.
4. valid = o < c compares them lexicographically, so the 18:00-09:00 Friday range prints INVALID.

Try it:

1. Detect overnight hours such as 22:00-02:00 and treat them as valid rather than inverted.
2. Parse the times into minutes and reject malformed strings that lack a colon.
3. Support split shifts where a day has a morning and an afternoon range.
4. Support an overnight shift by allowing close < open when a wrap flag is set.

⚠ Common Mistakes

- String comparison only works because times are zero-padded; '9:00' would sort wrong against '17:00'.
- Treating 'closed' as a range to split raises a ValueError without the early `continue`.
- An overnight span like 18:00-09:00 is flagged invalid here, which is correct unless you model wraparound.

Skills gained hours validation string splitting range checking

→ Where You'll Use This: Hours validation catches data-entry errors that mislead customers and search engines, a routine check when an SEO imports or audits opening times across many local listings.

Mental Model

Like checking a shift roster for a clock-out written before the clock-in, an obvious mistake that the comparison catches instantly.

Project 419 Detect a Profile Closed All Week

The Idea Behind It

A profile where every day reads closed is almost always a mistake that quietly tanks visibility. This code builds a list of days whose hours are not closed, and when that open-days list is empty it prints a warning; otherwise it lists the open days. The all-closed example yields an open-day count of zero and raises the alarm.

What You'll Build & Learn

Detect a business marked closed on every day and warn when no open days remain.

- Filtering a dict by value
- Truthiness of an empty list
- Branching on a guard condition

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 418: parse and validate opening hours. Project 420 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a profile closed all week is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `hours = {'mon': 'closed', 'tue': 'closed', 'wed': 'closed',` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.

- Evidence: `print('open day count:', len(open_days))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`hours = {'mon': 'closed', 'tue': 'closed', 'wed': 'closed',`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`open_days = [d for d, h in hours.items() if h != 'closed']`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('open day count:', len(open_days))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `open_days = [d for d, h in hours.items() if h != 'closed']`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Profile Closed All Week: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: hours, mon, closed, tue, wed. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: hours = {'mon': 'closed', 'tue': 'closed', 'wed': 'closed', | Core: open_days = [d for d, h in hours.items() if h != 'closed'] | Evidence: print('open day count:', len(open_days))

🛠️ Reader to Writer Bridge

1. Read: explain detect a profile closed all week in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 418.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 420 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('open day count:', len(open_days))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `hours = {'mon': 'closed', 'tue': 'closed', 'wed': 'closed',` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a profile closed all week and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Filtering the days whose value is not 'closed' yields the open days; an empty result means the business looks permanently closed and needs review.

```
hours = {'mon': 'closed', 'tue': 'closed', 'wed': 'closed',
         'thu': 'closed', 'fri': 'closed', 'sat': 'closed', 'sun': 'closed'}
open_days = [d for d, h in hours.items() if h != 'closed']
if not open_days:
    print('WARNING: business marked closed all week')
```

```
else:
    print('open days:', open_days)
print('open day count:', len(open_days))
```

```
output
WARNING: business marked closed all week
open day count: 0
```

How it works:

1. hours marks all seven days as closed.
2. open_days collects days where the value is not closed, leaving the list empty here.
3. if not open_days is True for the empty list, so the WARNING about being closed all week prints.
4. The final line reports an open-day count of zero.

Try it:

1. Warn when fewer than a minimum number of days are open, not only zero.
2. List the closed days when at least one day is open as a softer notice.
3. Distinguish a permanently closed marker from temporary holiday closures.
4. Open the business on one day and confirm the warning disappears.

⚠ Common Mistakes

- Checking `if open_days == []` works but `if not open_days` is the idiomatic empty test.
- Comparing to the string 'Closed' (capitalized) misses lowercase 'closed' values.
- Counting 'closed' days instead of open days inverts the logic.

Skills gained closed-week detection | value filtering | guard clause

→ Where You'll Use This: This guard catches a damaging configuration error during a profile audit, since an accidentally all-closed listing tells searchers the business is shut and suppresses it in local results.

Mental Model

Like spotting a shop whose sign reads closed every day of the week, a clear signal the sign itself is wrong.

Project 420 Audit Google Business Profile Fields

The Idea Behind It

A Google Business Profile has a known roster of fillable fields. This code lists the required ones, then builds a missing list of any field whose value is falsy, so an empty website string and a zero photos count both register

as gaps. Subtracting missing from the total and scaling to a percentage gives the 75 percent completeness at the heart of a GBP audit.

What You'll Build & Learn

Audit a Google Business Profile by listing empty required fields and scoring overall completeness.

- Defining a required-field list
- Detecting falsy values including zero
- Scoring completeness as a percentage

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 419: detect a profile closed all week. Project 421 changes the surface problem while preserving the underlying method.

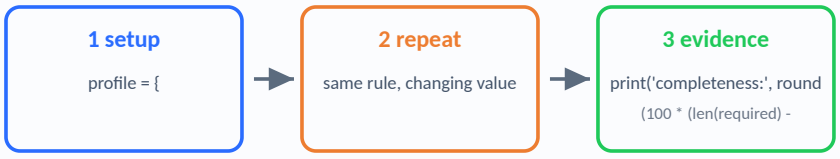
progress check

Reading target: audit google business profile fields is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `profile = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('completeness:', round(100 * (len(required) - len(missing)) / len(required), 1), '%')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`profile = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`missing = [f for f in required if not profile.get(f)]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('completeness:', round(100 * (len(required) - len(missing)) / len(required), 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [f for f in required if not profile.get(f)]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Audit Google Business Profile Fields: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: profile, name, Acme, Plumbing, address. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: profile = { | Core: missing = [f for f in required if not profile.get(f)] | Evidence: print('completeness:', round(100 * (len(required) - len(missing)) / len...

🛠️ Reader to Writer Bridge

1. Read: explain audit google business profile fields in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 419.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 421 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('completeness:', round(100 * (len(required) - len(missing)) / len(required), 1), '%')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `profile = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of audit google business profile fields and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Iterating the required fields and collecting those whose value is falsy produces a missing-field list and a completeness percentage in one pass.

```
profile = {
    'name': 'Acme Plumbing',
    'address': '123 Main St',
    'phone': '4155550199',
    'website': '',
    'hours': {'mon': '9–5'},
    'primary_category': 'Plumber',
    'description': 'Trusted plumbing.',
    'photos': 0,
}
required = ['name', 'address', 'phone', 'website', 'hours',
            'primary_category', 'description', 'photos']
missing = [f for f in required if not profile.get(f)]
print('GBP audit')
print('missing fields:', missing)
print('completeness:', round(100 * (len(required) - len(missing)) /
len(required), 1), '%')
```

```
output
GBP audit
missing fields: ['website', 'photos']
completeness: 75.0 %
```

How it works:

1. profile holds eight fields, with website an empty string and photos a zero.
2. required names the eight fields the audit expects to be filled.
3. missing keeps fields where profile.get(f) is falsy, capturing website and photos.
4. The completeness print scales required minus missing over required to 75.0 percent.

Try it:

1. Separate truly required fields from optional ones and score them differently.
2. Treat a photos count below three as incomplete rather than only zero.
3. Emit the audit as a structured dict suitable for a report rather than prints.
4. Add a 'service_area' field and mark it missing.

⚠ Common Mistakes

- An empty string website and zero photos are both falsy and correctly flagged as missing.
- Using `profile[f]` instead of `profile.get(f)` raises if a field is absent entirely.
- Counting present as `len(required) - len(missing)` assumes no field appears twice in the list.

Skills gained GBP audit | falsy detection | completeness percentage

→ Where You'll Use This: GBP audits are bread-and-butter local-SEO deliverables, where the list of empty fields and a completeness score become the action items handed to a client to strengthen their profile.

⚡ Mental Model

Like a building inspectors clipboard: every unchecked box on the required list is a defect the owner must remedy before approval.

Project 421 Detect a Map Embed on the Contact Page

💡 The Idea Behind It

Embedding a Google map on a contact page reinforces location relevance. This code runs a regex over the HTML to capture every iframe src, then checks with any whether one contains maps. The single iframe pointing at a google.com/maps/embed URL makes has_map True, and the source is printed truncated to its first forty characters.

What You'll Build & Learn

Find iframe sources in a contact page and confirm whether any of them embeds a Google map.

- Extracting attributes with re.findall
- Testing a condition across items with any
- Slicing a string to truncate output

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 420: audit google business profile fields. Project 422 changes the surface problem while preserving the underlying method.

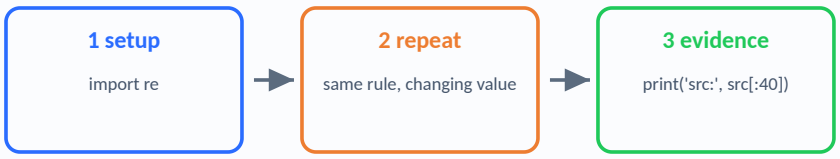
progress check

Reading target: detect a map embed on the contact page is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('src:', src[:40])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check

Move	Code to inspect	Why it matters
		nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`has_map = any('maps' in src for src in iframes)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('src:', src[:40])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `has_map = any('maps' in src for src in iframes)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Map Embed on the Contact Page: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: re, page, section, h2, Find. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: has_map = any('maps' in src for src in iframes) | Evidence: print('src:', src[:40])

Reader to Writer Bridge

1. Read: explain detect a map embed on the contact page in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 420.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 422 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('src:', src[:40])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a map embed on the contact page and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A non-greedy regex captures each iframe's `src`, and an `any(...)` test for 'maps' in those sources confirms a map embed is present without a full HTML parse.

```
import re
page = '''<section><h2>Find Us</h2>
<iframe src="https://www.google.com/maps/embed?pb=abc" width="600"></iframe>
</section>'''
iframes = re.findall(r'<iframe[^>]*src="([^"]+)"', page)
has_map = any('maps' in src for src in iframes)
print('iframes found:', len(iframes))
print('map embed present:', has_map)
for src in iframes:
    print('src:', src[:40])
```

```
output
iframes found: 1
map embed present: True
src: https://www.google.com/maps/embed?pb=abc
```

How it works:

1. page holds a snippet of contact-page HTML containing one iframe with a maps src.
2. re.findall with the iframe src pattern captures the list of source URLs.
3. any('maps' in src for src in iframes) returns True because the URL contains maps.
4. The loop prints each src truncated to forty characters with src[:40].

Try it:

1. Match map embeds from other providers such as OpenStreetMap or Bing Maps.

2. Count how many distinct iframes appear and warn if there are none.
3. Extract the embed coordinates from the src and verify they match the business.
4. Add a second non-map iframe and confirm only the map is detected.

⚠ Common Mistakes

- A greedy `.*` in the src pattern would swallow past the closing quote on multi-attribute tags.
- Matching only lowercase 'maps' misses a capitalized host; normalize if needed.
- Assuming the first iframe is the map; a page may embed several iframes.

Skills gained map embed detection | regex extraction | any check

→ Where You'll Use This: Detecting a map embed is part of an on-page local audit, where an SEO confirms the contact page reinforces the businesss location to both visitors and search engines.

Mental Model

Like scanning a brochure for the little inset map, confirming the customer is shown exactly where to find the door.

Project 422 Check LocalBusiness Schema Completeness

The Idea Behind It

Rich results depend on complete LocalBusiness structured data. This code parses a JSON-LD string with json.loads, defines the properties a strong markup should carry, then builds a sorted list of any required property absent from the parsed object. With geo and priceRange missing, the complete flag reads False, auditing the markup directly.

What You'll Build & Learn

Parse LocalBusiness JSON-LD and report which required schema properties are missing.

- Parsing JSON-LD with json.loads
- Checking key presence in a dict
- Listing missing required properties

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 421: detect a map embed on the contact page. Project 423 changes the surface problem while preserving the underlying method.

progress check

Reading target: check localbusiness schema completeness is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('complete:', not missing)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`missing = sorted(f for f in required if f not in schema)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('complete:', not missing)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = sorted(f for f in required if f not in schema)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Check LocalBusiness Schema Completeness: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: json, schema, loads, type, LocalBusiness. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import json | Core: missing = sorted(f for f in required if f not in schema) | Evidence: print('complete:', not missing)

🛠️ Reader to Writer Bridge

1. Read: explain check localbusiness schema completeness in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 421.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 423 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('complete:', not missing)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of check localbusiness schema completeness and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Loading the JSON-LD with `json.loads` and diffing its keys against a required-property list surfaces exactly which schema fields are missing.

```
import json
schema = json.loads('''{
  "@type": "LocalBusiness",
  "name": "Acme Plumbing",
  "telephone": "+14155550199",
  "address": {"streetAddress": "123 Main St", "postalCode": "94110"},
  "openingHours": "Mo-Fr 09:00-17:00"
}''')
required = ['name', 'address', 'telephone', 'openingHours', 'priceRange',
'geo']
missing = sorted(f for f in required if f not in schema)
print('type:', schema['@type'])
print('missing schema props:', missing)
print('complete:', not missing)
```

```
output
type: LocalBusiness
missing schema props: ['geo', 'priceRange']
complete: False
```

How it works:

1. json.loads turns the JSON-LD string into a dict named schema.
2. required lists six properties a complete LocalBusiness markup should include.
3. missing keeps required properties absent from schema, sorted, yielding geo and priceRange.
4. The prints show the schema type, the missing props, and complete as False because the list is non-empty.

Try it:

1. Validate that the address sub-object itself contains required keys like streetAddress.
2. Check the value types so telephone is a string and geo holds coordinates.
3. Load several JSON-LD blocks from a page and audit each in turn.
4. Add 'geo' and 'priceRange' to the JSON and watch it report complete.

⚠ Common Mistakes

- Checking for nested properties like 'address.streetAddress' needs a deeper walk, not a top-level key test.

- Treating a present-but-empty property as complete; this checks key presence only.
- Forgetting to sort the missing list makes the output order nondeterministic.

Skills gained schema validation JSON-LD parsing property check

→ **Where You'll Use This:** Structured-data audits feed an SEOs technical recommendations, since complete LocalBusiness markup with geo, priceRange, and hours unlocks the rich features that lift local listings in search.

Mental Model

Like inspecting a passport for every required stamp before letting it through customs, noting exactly which pages are blank.

Project 423 Measure Citation Source Diversity

The Idea Behind It

Links from many kinds of sources beat a pile from one kind. This code tags each citation with a type, then uses Counter to tally how many fall into review, directory, and authority buckets. Printing the count per sorted type and the number of distinct types measures source diversity rather than raw citation volume.

What You'll Build & Learn

Measure citation source diversity by counting how many citations fall into each distinct type.

- Tallying categories with Counter
- Iterating sorted dictionary keys
- Counting distinct keys with len

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 422: check localbusiness schema completeness. Project 424 changes the surface problem while preserving the underlying method.

progress check

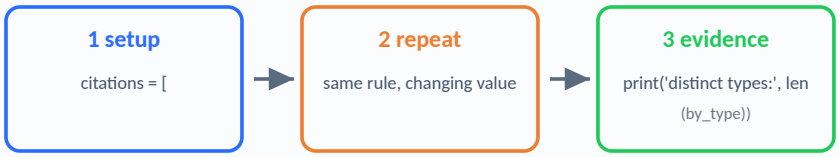
Reading target: measure citation source diversity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `citations = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('distinct types:', len(by_type))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`citations = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`by_type = Counter(c['type'] for c in citations)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('distinct types:', len(by_type))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `by_type = Counter(c['type'] for c in citations)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure Citation Source Diversity: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: citations, source, Yelp, type, review. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: citations = [| Core: by_type = Counter(c['type'] for c in citations) | Evidence: print('distinct types:', len(by_type))

🛠️ Reader to Writer Bridge

1. Read: explain measure citation source diversity in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 422.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 424 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('distinct types:', len(by_type))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `citations = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure citation source diversity and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A `Counter` over the citation types tallies each bucket, and the number of distinct types is a simple diversity score for the backlink and citation profile.

```
citations = [
    {'source': 'Yelp', 'type': 'review'},
    {'source': 'YellowPages', 'type': 'directory'},
    {'source': 'BBB', 'type': 'authority'},
    {'source': 'Foursquare', 'type': 'review'},
    {'source': 'LocalChamber', 'type': 'authority'},
]
from collections import Counter
by_type = Counter(c['type'] for c in citations)
print('total citations:', len(citations))
for t in sorted(by_type):
    print(f'  {t}: {by_type[t]}')
print('distinct types:', len(by_type))
```

```
output
total citations: 5
  authority: 2
  directory: 1
  review: 2
distinct types: 3
```

How it works:

1. citations is a list of dicts each with a source and a type such as review or authority.
2. Counter over c['type'] tallies how many citations belong to each type.
3. The loop prints each type in sorted order with its count.
4. len(by_type) reports three distinct types across the five citations.

Try it:

1. Flag a type that dominates the mix beyond a chosen share of the total.
2. Recommend which under-represented type to pursue next based on the counts.
3. Add a domain field and count distinct domains alongside distinct types.
4. Add a 'social' type citation and watch diversity rise.

⚠ Common Mistakes

- Counting total citations alone misses that they might all be the same low-value type.
- Iterating the Counter unsorted prints types in a nondeterministic order.
- Treating two listings from the same source as separate types inflates diversity.

Skills gained diversity measurement | Counter tallying | type counting

→ Where You'll Use This: Source diversity guides link-building strategy, prompting an SEO to chase authority and directory citations rather than piling up more reviews-only sources for diminishing returns.

⚡ Mental Model

Like judging a diet by variety, not bulk: five vegetables beat five servings of the same potato for overall health.

Project 424 Separate Primary from Secondary Categories

💡 The Idea Behind It

Exactly one category should be primary while the rest stay secondary. This code reads a list of category dicts carrying a primary flag, splits the names into primary and secondary lists, and only prints a clean primary when precisely one exists, otherwise marking the count INVALID. Here Plumber is the lone primary with two secondaries listed cleanly.

What You'll Build & Learn

Validate that a profile has exactly one primary category and list the secondaries.

- Splitting items by a boolean flag
- Validating a single-element expectation
- Conditional print on a list length

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 423: measure citation source diversity. Project 425 changes the surface problem while preserving the underlying method.

progress check

Reading target: separate primary from secondary categories is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

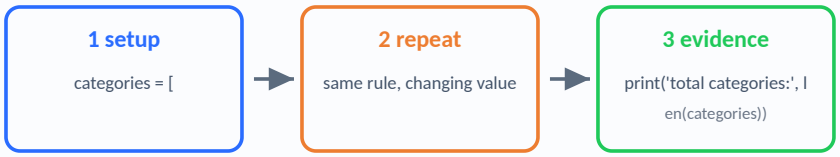
- Setup: `categories = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('total categories:', len(categories))` exposes the result; match it against the output before changing the program.

output proves the trace

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`categories = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`primary = [c['name'] for c in categories if c['primary']]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('total categories:', len(categories))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `primary = [c['name'] for c in categories if c['primary']]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Separate Primary from Secondary Categories: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.

Layer	What to read
Names	Watch these names first: categories, name, Plumber, primary, Water. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: categories = [| Core: primary = [c['name'] for c in categories if c['primary']] | Evidence: print('total categories:', len(categories))

🛠️ Reader to Writer Bridge

1. Read: explain separate primary from secondary categories in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 423.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 425 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('total categories:', len(categories))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `categories = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of separate primary from secondary categories and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Partitioning categories on their `primary` flag isolates the one primary (flagging the count if not exactly one) and collects the secondary list.

```
categories = [
    {'name': 'Plumber', 'primary': True},
    {'name': 'Water Heater Installation', 'primary': False},
    {'name': 'Drain Cleaning Service', 'primary': False},
]
primary = [c['name'] for c in categories if c['primary']]
secondary = [c['name'] for c in categories if not c['primary']]
print('primary:', primary[0] if len(primary) == 1 else f'INVALID
({len(primary)})')
```

```
print('secondary:', secondary)
print('total categories:', len(categories))
```

```
output
primary: Plumber
secondary: ['Water Heater Installation', 'Drain Cleaning Service']
total categories: 3
```

How it works:

1. categories is a list of dicts each with a name and a primary boolean.
2. primary collects names where the flag is True; secondary collects the rest.
3. The print emits primary[0] only when exactly one primary exists, else INVALID with the count.
4. secondary prints the two non-primary categories and total prints the count of three.

Try it:

1. Warn explicitly when zero primaries are set versus when several are.
2. Cap the number of secondary categories at a recommended maximum.
3. Promote the first category to primary automatically when none is flagged.
4. Mark a second category primary and watch the INVALID count appear.

⚠ Common Mistakes

- Assuming `primary[0]` exists crashes when no category is marked primary.
- Allowing two primaries silently picks the first instead of flagging the conflict.
- Sorting the secondary list would change the intended display order.

Skills gained category validation | flag filtering | single-primary check

→ Where You'll Use This: Enforcing a single primary category matches how Google treats profiles, so an SEO can flag a misconfigured listing that names two primaries or none before it muddies ranking relevance.

⚡ Mental Model

Like a band needing one lead singer: backing vocalists are fine in number, but two front-and-center leads is a problem.

Project 425 Match Website NAP Against the Profile

💡 The Idea Behind It

Your own website contact details must agree with your Google profile. This code normalizes the sites formatted phone to bare digits, merges that into a copy of the website NAP, then diffs every field against the profile NAP. With the phone cleaned to match, the mismatch list is empty, confirming an in-sync footer.

What You'll Build & Learn

Match a website's NAP against the Google profile by normalizing the phone and diffing the fields.

- Stripping non-digits from a phone
- Merging dicts with the spread operator
- Diffing fields for mismatches

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 424: separate primary from secondary categories. Project 426 changes the surface problem while preserving the underlying method.

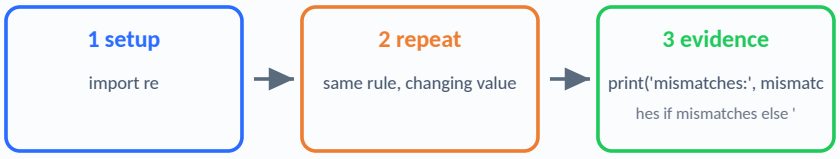
progress check

Reading target: match website nap against the profile is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('mismatches:', mismatches if mismatches else 'none')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`return ''.join(ch for ch in p if ch.isdigit())`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('mismatches:', mismatches if mismatches else 'none')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return ''.join(ch for ch in p if ch.isdigit())`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Match Website NAP Against the Profile: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: re, website_nap, name, acme, plumbing. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: return ''.join(ch for ch in p if ch.isdigit()) | Evidence: print('mismatches:', mismatches if mismatches else 'none')

🛠️ Reader to Writer Bridge

1. Read: explain match website nap against the profile in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 424.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 426 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('mismatches:', mismatches if mismatches else 'none')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of match website nap against the profile and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Cleaning the website phone to digits before comparison removes formatting noise so only genuine NAP discrepancies between site and profile are reported.

```
import re
website_nap = {
    'name': 'acme plumbing',
    'phone': '(415) 555-0199',
    'zip': '94110',
}
profile_nap = {'name': 'acme plumbing', 'phone': '4155550199', 'zip': '94110'}
def clean_phone(p):
    return ''.join(ch for ch in p if ch.isdigit())
norm_site = {**website_nap, 'phone': clean_phone(website_nap['phone'])}
mismatches = sorted(f for f in profile_nap if norm_site[f] != profile_nap[f])
print('website phone normalized:', norm_site['phone'])
print('mismatches:', mismatches if mismatches else 'none')
```

```
output
```

```
website phone normalized: 4155550199
mismatches: none
```

How it works:

1. website_nap holds a formatted phone while profile_nap holds the bare-digit version.
2. clean_phone keeps only digit characters from the website phone.
3. norm_site copies website_nap with the cleaned phone substituted via dict unpacking.
4. mismatches collects fields where norm_site differs from profile_nap, printing none after the phone is normalized.

Try it:

1. Extract the NAP from real page HTML instead of a hand-built dict.
2. Normalize the name and zip too so only true differences flag.
3. Report the specific old and new values for each mismatched field.
4. Change the site zip to introduce a real mismatch.

⚠ Common Mistakes

- Comparing the formatted site phone '(415) 555-0199' to the digit profile phone reports a false mismatch.
- Mutating `website_nap` in place instead of building `norm_site` would corrupt the original record.
- Not sorting the mismatch list makes the message nondeterministic.

Skills gained website NAP match | phone cleaning | field diffing

→ Where You'll Use This: Reconciling on-page NAP with the profile catches an out-of-sync footer, a common technical-SEO fix that keeps the businesss own site from contradicting its directory listings.

⚡ Mental Model

Like checking that the address on your business card matches the one on your storefront door before either misleads a customer.

Project 426 Flag a Keyword-Stuffed Business Name

The Idea Behind It

Stuffing a service keyword into your business name breaks Google's rules and risks suspension. This code scans each name for an embedded keyword by lowercasing and testing membership, labeling matches keyword-stuffed and the rest clean. San Francisco Plumber Co trips the check while Acme Plumbing and Bobs Cafe pass, and the offender count totals one.

What You'll Build & Learn

Flag business names that embed a service keyword and count how many names are stuffed.

- Case-insensitive substring search
- Labeling items by a condition
- Counting matches with sum

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 425: match website nap against the profile. Project 427 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag a keyword-stuffed business name is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `businesses = ['Acme Plumbing', 'San Francisco Plumber Co', 'Bobs Cafe']` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('stuffed names:', stuffed)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`businesses = ['Acme Plumbing', 'San Francisco Plumber Co', 'Bobs Cafe']`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.

Move	Code to inspect	Why it matters
2. Apply the rule	`note = 'keyword-stuffed name' if flag else 'clean'`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('stuffed names:', stuffed)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `note = 'keyword-stuffed name' if flag else 'clean'`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag a Keyword-Stuffed Business Name: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: businesses, Acme, Plumbing, San, Francisco. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: businesses = ['Acme Plumbing', 'San Francisco Plumber Co', 'Bobs Cafe'] | Core: note = 'keyword-stuffed name' if flag else 'clean' | Evidence: print('stuffed names:', stuffed)

Reader to Writer Bridge

1. Read: explain flag a keyword-stuffed business name in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 425.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 427 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('stuffed names:', stuffed)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `businesses = ['Acme Plumbing', 'San Francisco Plumber Co', 'Bobs Cafe']` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag a keyword-stuffed business name and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A case-insensitive substring test for the keyword inside each name flags listings that stuff a service term into the brand name.

```
businesses = ['Acme Plumbing', 'San Francisco Plumber Co', 'Bobs Cafe']
keyword = 'plumber'
for name in businesses:
    flag = keyword in name.lower()
    note = 'keyword-stuffed name' if flag else 'clean'
    print(f'{name}: {note}')
stuffed = sum(1 for n in businesses if keyword in n.lower())
print('stuffed names:', stuffed)
```

```
output
Acme Plumbing: clean
San Francisco Plumber Co: keyword-stuffed name
Bobs Cafe: clean
stuffed names: 1
```

How it works:

1. businesses lists three names and keyword holds the service term plumber.
2. The loop tests keyword in name.lower(), labeling a hit keyword-stuffed name and otherwise clean.
3. San Francisco Plumber Co contains plumber so it is flagged, while the others pass.
4. stuffed sums the flagged names, printing a count of one.

Try it:

1. Check against a list of many service keywords rather than a single one.
2. Use word-boundary matching so plumber inside plumbery does not false-positive.
3. Report the matched keyword alongside each flagged name.
4. Use a word-boundary regex so 'Plumberton' is not flagged.

⚠ Common Mistakes

- Case-sensitive matching misses 'Plumber' when you search for 'plumber'.
- A substring test flags 'Plumberton' (a place) as stuffed; word-boundary matching is stricter.
- Counting flagged names without listing them hides which need a rename.

Skills gained name spam check | substring search | violation counting

→ Where You'll Use This: Keyword-stuffed-name detection guards against a guideline violation that can suspend a listing, a compliance scan an SEO runs before submitting or auditing business names.

⚡ Mental Model

Like a referee watching for a player who sneaks an advertisement onto their jersey: the rule is clear and the violation stands out.

Project 427 Rank Competitors by Distance

💡 The Idea Behind It

Knowing which competitors sit nearest a searcher explains the local pack you fight in. This code skips any library, computing a plain Euclidean distance on raw coordinate pairs with the difference of squares under a square root. Sorting competitors by that distance and enumerating the result ranks PipeKing closest and FastDrain farthest from your location.

What You'll Build & Learn

Rank competitors by plain Euclidean distance on their coordinates relative to your own position.

- Euclidean distance on coordinate pairs
- Rounding a computed distance
- Sorting then enumerating ranks

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 426: flag a keyword-stuffed business name. Project 428 changes the surface problem while preserving the underlying method.

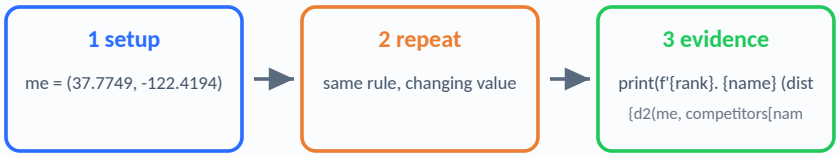
progress check

Reading target: rank competitors by distance is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `me = (37.7749, -122.4194)` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print(f'{rank}. {name} (dist {d2(me, competitors[name])})')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`me = (37.7749, -122.4194)`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`for rank, name in enumerate(ranked, 1):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{rank}. {name} (dist {d2(me, competitors[name])})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for rank, name in enumerate(ranked, 1):`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Rank Competitors by Distance: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: me, competitors, PlumbPro, FastDrain, PipeKing. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: me = (37.7749, -122.4194) | Core: for rank, name in enumerate(ranked, 1): | Evidence: print(f'{rank}. {name} (dist {d2(me, competitors[name])})')

Reader to Writer Bridge

1. Read: explain rank competitors by distance in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 426.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 428 should feel like a variation, not a brand-new mystery.

   from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{rank}. {name} (dist {d2(me, competitors[name])})')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `me = (37.7749, -122.4194)` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of rank competitors by distance and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Computing the straight-line distance in degree space and sorting by it ranks competitors by nearness; degrees suffice for ordering even though they are not kilometres.

```
me = (37.7749, -122.4194)
competitors = {
    'PlumbPro': (37.7849, -122.4094),
    'FastDrain': (37.7600, -122.4500),
    'PipeKing': (37.7750, -122.4180),
}
def d2(a, b):
    return round(((a[0]-b[0])**2 + (a[1]-b[1])**2) ** 0.5, 4)
ranked = sorted(competitors, key=lambda n: d2(me, competitors[n]))
for rank, name in enumerate(ranked, 1):
    print(f'{rank}. {name} (dist {d2(me, competitors[name])})')
```

```
output
1. PipeKing (dist 0.0014)
2. PlumbPro (dist 0.0141)
3. FastDrain (dist 0.034)
```

How it works:

1. me holds your coordinates and competitors maps three rivals to their own.
2. d2 computes the square root of summed squared coordinate differences, rounded to four places.
3. sorted orders competitors by d2 from me, putting PipeKing first.
4. enumerate starting at one prints the ranked list with each rivals distance.

Try it:

1. Swap in the kilometre-accurate scaling from the earlier proximity project.
2. Include your own business in the list and show where it ranks.
3. Filter the competitor set to those within a chosen distance before ranking.
4. Add a competitor at your exact coordinates and confirm it ranks first.

⚠ Common Mistakes

- Comparing raw degree distance is fine for ranking but must not be reported as kilometres.
- Forgetting the square root still ranks correctly but the printed values are squared distances.
- Rounding before sorting could tie two competitors that the full precision separates.

Skills gained `competitor ranking` `euclidean distance` `rank enumeration`

→ Where You'll Use This: Competitor proximity ranking sharpens local competitive analysis, showing an SEO which nearby rivals share a searchers map pack and deserve the closest watch.

⚡ Mental Model

Like lining up runners by how close they stand to the finish line before the race even starts.

Project 428 Flag a Spam Listing

💡 The Idea Behind It

Spam profiles share telltale signs you can tally into a score. This code checks four flags: a name heavy with digits and symbols, a fake phone of repeated digits, a same-day review burst above ten, and a missing address. Each triggered flag joins a list, and the example listing trips all four for a maximum spam score of four.

What You'll Build & Learn

Score how spammy a listing looks by tallying flags for promotional names, fake phones, review bursts, and missing data.

- Counting digit and symbol characters
- Detecting a single repeated digit
- Accumulating flags into a list

🧭 Skill Roadmap

`Skill step:` pushes the same idea into a less automatic situation. It builds on Project 427: rank competitors by distance. Project 429 changes the surface problem while preserving the underlying method. *progress check*

`Reading target:` flag a spam listing is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `listing = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('spam score:', len(flags))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`listing = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`if sum(c.isdigit() or c in '#!' for c in listing['name']) > 3:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('spam score:', len(flags))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if sum(c.isdigit() or c in '#!' for c in listing['name']) > 3:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag a Spam Listing: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: listing, name, Best, Cheap, Plumber. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: listing = { | Core: if sum(c.isdigit() or c in '#!' for c in listing['name']) > 3: | Evidence: print('spam score:', len(flags))

🛠️ Reader to Writer Bridge

1. Read: explain flag a spam listing in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 427.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 429 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('spam score:', len(flags))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `listing = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of flag a spam listing and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Each heuristic appends a named flag, and the flag count becomes a spam score you can threshold to triage suspicious listings.

```
listing = {
    'name': 'Best #1 Cheap Plumber Near Me 24/7',
    'phone': '0000000000',
    'reviews_same_day': 18,
    'address': '',
}
flags = []
if sum(c.isdigit() or c in '#!' for c in listing['name']) > 3:
    flags.append('promotional name')
if listing['phone'] == '0000000000' or len(set(listing['phone'])) == 1:
    flags.append('fake phone')
if listing['reviews_same_day'] > 10:
    flags.append('review burst')
if not listing['address']:
    flags.append('no address')
print('spam flags:', flags)
print('spam score:', len(flags))
```

```
output
spam flags: ['promotional name', 'fake phone', 'review burst', 'no address']
spam score: 4
```

How it works:

1. listing holds a promotional name, a zeroed phone, a same-day review burst, and an empty address.
2. The name flag counts characters that are digits or hash and bang, firing when more than three appear.
3. The phone flag fires when it equals all zeros or len(set) is one, meaning a single repeated digit.
4. Burst and address flags add when same-day reviews exceed ten or the address is empty, totaling a spam score of four.

Try it:

1. Add a flag for a name in all capital letters or excessive emoji.
2. Weight the flags so a fake phone counts more than a busy review day.
3. Return the triggered flags with a severity label rather than only a count.
4. Add a flag for a name longer than 50 characters.

⚠ Common Mistakes

- Counting only digits for the promotional-name test misses symbol stuffing like '#1' and '!!!'.
- A phone of all identical digits is caught by `len(set(phone)) == 1`; a real number rarely is.
- A high same-day review count alone is suggestive, not proof; weigh it with other flags.

Skills gained spam scoring | heuristic flags | character counting

→ Where You'll Use This: Spam scoring helps an SEO report fake or manipulative competitors to Google, where stacking several heuristic flags builds a credible case that a listing violates guidelines.

⚡ Mental Model

Like a fraud detector adding up red flags on an application: no single one convicts, but four together paint a clear picture.

Project 429 Audit Photo-Type Coverage

💡 The Idea Behind It

A complete profile shows exterior, interior, team, logo, and product photos. This code counts total images, then builds a missing list of photo types whose count is zero, here interior and product. Dividing the covered types by the expected five gives a 60 percent coverage figure that reveals gaps in visual completeness.

What You'll Build & Learn

Audit photo coverage by listing photo types with zero images and scoring overall visual completeness.

- Summing dict values for a total
- Finding keys with a zero value
- Scoring coverage as a percentage

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 428: flag a spam listing. Project 430 changes the surface problem while preserving the underlying method. *progress check*

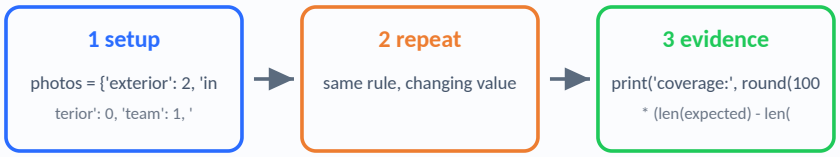
Reading target: audit photo-type coverage is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `photos = {'exterior': 2, 'interior': 0, 'team': 1, 'logo': 1, 'product': 0}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as data -> transformation -> evidence, not as isolated library calls.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('coverage:', round(100 * (len(expected) - len(missing)) / len(expected), 1), '%')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`photos = {'exterior': 2, 'interior': 0, 'team': 1, 'logo': 1, 'product': 0}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`missing = [k for k in expected if photos.get(k, 0) == 0]`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print('coverage:', round(100 * (len(expected) - len(missing)) / len(expected), 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `missing = [k for k in expected if photos.get(k, 0) == 0]`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Audit Photo-Type Coverage: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: photos, exterior, interior, team, logo. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: photos = {'exterior': 2, 'interior': 0, 'team': 1, 'logo': 1, 'product'... | Core: missing = [k for k in expected if photos.get(k, 0) == 0] | Evidence: print('coverage:', round(100 * (len(expected) - len(missing)) / len(exp...

🛠️ Reader to Writer Bridge

1. Read: explain audit photo-type coverage in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 428.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 430 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('coverage:', round(100 * (len(expected) - len(missing)) / len(expected), 1), '%')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `photos = {'exterior': 2, 'interior': 0, 'team': 1, 'logo': 1, 'product': 0}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of audit photo-type coverage and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering the expected photo types to those with a zero count lists the missing categories and yields a coverage percentage over the expected set.

```
photos = {'exterior': 2, 'interior': 0, 'team': 1, 'logo': 1, 'product': 0}
expected = ['exterior', 'interior', 'team', 'logo', 'product']
total = sum(photos.values())
missing = [k for k in expected if photos.get(k, 0) == 0]
print('total photos:', total)
print('missing photo types:', missing)
print('coverage:', round(100 * (len(expected) - len(missing)) / len(expected),
1), '%')
```

```
output
total photos: 4
missing photo types: ['interior', 'product']
coverage: 60.0 %
```

How it works:

1. photos maps five photo types to their image counts, with interior and product at zero.
2. total sums all the counts to four images.
3. missing keeps expected types whose count is zero via photos.get(k, 0), capturing interior and product.
4. The coverage print scales expected minus missing over expected to 60.0 percent.

Try it:

1. Require a minimum count per type so one product photo still counts as thin.
2. Add categories like food or video and recompute coverage.
3. Rank the missing types by which most influences engagement.
4. Add an 'interior' photo and watch coverage rise.

⚠ Common Mistakes

- Summing the values gives a total photo count, not how many types are covered.
- Using `photos[k]` instead of `photos.get(k, 0)` raises if an expected type is absent.
- Counting a type with one photo the same as one with ten ignores depth of coverage.

Skills gained photo audit zero-value detection coverage scoring

→ Where You'll Use This: Photo-coverage audits translate into concrete client tasks, since profiles rich in exterior, interior, and team imagery earn more engagement and outrank visually bare competitors in local results.

Mental Model

Like checking a real-estate listing for the missing room shots: an album with no interior photos leaves buyers guessing.

Project 430 Track a Rating Trend and Alert

The Idea Behind It

A falling star rating is an early warning worth catching fast. This code compares a prior average to the current one, computes the signed delta, derives the new-review count from the two totals, and labels the direction dropping or rising. A sensitivity threshold of 0.3 turns the 0.4 drop into a True alert, formatted neatly with a sign.

What You'll Build & Learn

Track a rating trend between two snapshots and decide whether the change warrants an alert.

- Computing a signed delta
- Deriving new reviews from two counts
- Alerting on an absolute threshold

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 429: audit photo-type coverage. Project 431 changes the surface problem while preserving the underlying method. *progress check*

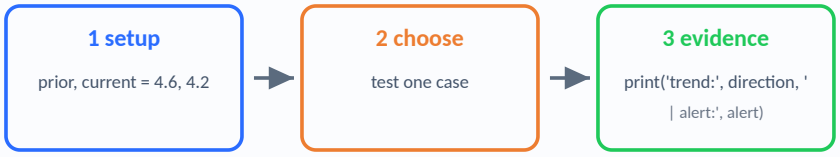
Reading target: track a rating trend and alert is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `prior, current = 4.6, 4.2` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.

- Evidence: `print('trend:', direction, '| alert:', alert)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`prior, current = 4.6, 4.2`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`direction = 'dropping' if delta < 0 else 'rising'`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('trend:', direction, '| alert:', alert)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `direction = 'dropping' if delta < 0 else 'rising'`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Track a Rating Trend and Alert: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: prior, current, prior_n, current_n, new_reviews. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: prior, current = 4.6, 4.2 | Core: direction = 'dropping' if delta < 0 else 'rising' | Evidence: print('trend:', direction, '| alert:', alert)

🛠️ Reader to Writer Bridge

1. Read: explain track a rating trend and alert in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 429.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 431 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('trend:', direction, '| alert:', alert)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `prior, current = 4.6, 4.2` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of track a rating trend and alert and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Subtracting the prior rating from the current gives a signed delta; comparing its magnitude to a threshold turns a small float change into an actionable alert flag.

```
prior, current = 4.6, 4.2
prior_n, current_n = 40, 52
new_reviews = current_n - prior_n
delta = round(current - prior, 2)
direction = 'dropping' if delta < 0 else 'rising'
```

```
alert = abs(delta) >= 0.3
print(f'rating: {prior} -> {current} ({"+" if delta>=0 else ""}{delta})')
print('new reviews:', new_reviews)
print('trend:', direction, '| alert:', alert)
```

```
output
rating: 4.6 -> 4.2 (-0.4)
new reviews: 12
trend: dropping | alert: True
```

How it works:

1. prior and current hold the old and new averages, with their review counts in prior_n and current_n.
2. new_reviews subtracts the counts to twelve, and delta rounds current minus prior to negative 0.4.
3. direction reads dropping because delta is below zero.
4. alert is abs(delta) >= 0.3, which is True, and the f-string prints the signed change and verdict.

Try it:

1. Make the alert threshold configurable per client sensitivity.
2. Factor the new-review volume into the alert so a tiny sample is downweighted.
3. Track a rolling history of averages rather than just two points.
4. Raise the current rating above the prior and watch the trend flip to rising.

⚠ Common Mistakes

- Comparing the raw delta to the threshold without `abs` misses a sharp drop as 'negative, ignore'.
- Printing the unrounded delta shows float noise like -0.40000000000000036.
- Inferring direction from review count instead of the rating delta answers the wrong question.

Skills gained rating trend | delta computation | threshold alert

→ Where You'll Use This: Rating-trend alerts power reputation monitoring, where an SEO is notified the moment a clients average drops enough to threaten star snippets and click-through before reviews pile up.

Mental Model

Like a smoke alarm tuned to a sensitivity: a small dip is ignored, but a real drop sets it off before the fire spreads.

Project 431 Compute a Weighted Local-Ranking Score

The Idea Behind It

Local ranking blends several signals at different weights into one figure. This code pairs a weights dict summing to one with normalized signal strengths between zero and one, then computes a weighted sum across the shared keys. Printing each signal times its weight and scaling the total by a hundred yields a composite local score of 77 out of 100.

What You'll Build & Learn

Combine weighted, normalized local-ranking signals into a single composite score out of 100.

- Pairing weights with signal values
- Computing a weighted sum
- Scaling a normalized total to 100

Skill Roadmap

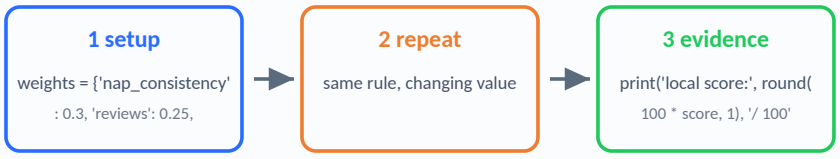
Skill step: pushes the same idea into a less automatic situation. It builds on Project 430: track a rating trend and alert. Project 432 changes the surface problem while preserving the underlying method. *progress check*

Reading target: compute a weighted local-ranking score is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `weights = {'nap_consistency': 0.3, 'reviews': 0.25, 'proximity': 0.2,` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `print('local score:', round(100 * score, 1), '/ 100')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`weights = {'nap_consistency': 0.3, 'reviews': 0.25, 'proximity': 0.2,`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`score = sum(weights[k] * signals[k] for k in weights)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('local score:', round(100 * score, 1), '/ 100')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `score = sum(weights[k] * signals[k] for k in weights)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute a Weighted Local-Ranking Score: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: weights, nap_consistency, reviews, proximity, profile_complete. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: weights = {'nap_consistency': 0.3, 'reviews': 0.25, 'proximity': 0.2, | Core: score = sum(weights[k] *signals[k] for k in weights)* | *Evidence: print('local score:', round(100* score, 1), '/ 100')

🛠️ Reader to Writer Bridge

1. Read: explain compute a weighted local-ranking score in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 430.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 432 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('local score:', round(100 * score, 1), '/ 100')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `weights = {'nap_consistency': 0.3, 'reviews': 0.25, 'proximity': 0.2,` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute a weighted local-ranking score and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Multiplying each normalized signal by its weight and summing gives a single 0-to-1 score; scaling by 100 makes a readable composite, with sorted keys keeping the breakdown deterministic.

```
weights = {'nap_consistency': 0.3, 'reviews': 0.25, 'proximity': 0.2,
           'profile_complete': 0.15, 'categories': 0.1}
signals = {'nap_consistency': 0.8, 'reviews': 0.6, 'proximity': 0.9,
           'profile_complete': 1.0, 'categories': 0.5}
score = sum(weights[k] * signals[k] for k in weights)
for k in sorted(weights):
    print(f'{k}: {signals[k]} x {weights[k]} = {round(weights[k]*signals[k],
3)}')
print('local score:', round(100 * score, 1), '/ 100')
```

```
output
categories: 0.5 x 0.1 = 0.05
nap_consistency: 0.8 x 0.3 = 0.24
profile_complete: 1.0 x 0.15 = 0.15
proximity: 0.9 x 0.2 = 0.18
```

```
reviews: 0.6 x 0.25 = 0.15
local score: 77.0 / 100
```

How it works:

1. weights assigns each of five signals a share that sums to one.
2. signals holds each factors normalized strength between zero and one.
3. score sums weights[k] times signals[k] across the shared keys.
4. The loop prints each contribution in sorted order and the final line scales score by a hundred to 77.0.

Try it:

1. Validate that the weights sum to one and warn otherwise.
2. Add a new signal such as citation diversity with its own weight.
3. Show which low-scoring signal drags the composite down the most.
4. Bump the reviews signal to 1.0 and watch the score climb.

⚠ Common Mistakes

- Weights that do not sum to 1 silently rescale the maximum away from 100.
- Iterating the weights dict unsorted prints the breakdown in a nondeterministic order.
- Multiplying a signal above 1.0 (out of range) would push the score past 100.

Skills gained weighted scoring | signal blending | composite metric

→ Where You'll Use This: A weighted composite score lets an agency roll many local signals into one dial a client understands, and tuning the weights models how Google might prioritize proximity, reviews, and consistency.

Mental Model

Like a final grade built from weighted exams and homework: each input pushes the number by exactly its assigned share.

Project 432 Run a Full Local-SEO Health Report

The Idea Behind It

This capstone ties the chapter together into a single audit. It computes NAP consistency from citations matching a canonical tuple, scores profile completeness from a fields dict, averages the two percentages, and prints a PASS or NEEDS WORK verdict against a seventy-percent bar. The example earns 73.3 percent and passes, the skeleton of a real local audit.

What You'll Build & Learn

Combine NAP consistency and profile completeness into one local-SEO health score with a pass or needs-work verdict.

- Reusing consistency and completeness logic
- Averaging two percentages
- Producing a pass or fail verdict

Skill Roadmap

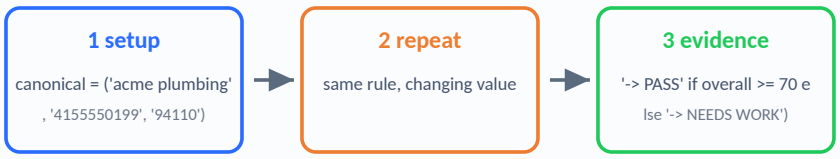
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 431: compute a weighted local-ranking score. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run a full local-seo health report is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `canonical = ('acme plumbing', '4155550199', '94110')` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: check nap consistency across citations, score local signals, and audit business-profile completeness.
- Evidence: `'-> PASS' if overall >= 70 else '-> NEEDS WORK')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`canonical = ('acme plumbing', '4155550199', '94110')`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: check

Move	Code to inspect	Why it matters
		nap consistency across citations, score local signals, and audit business-profile completeness.
2. Apply the rule	`consistent = sum(1 for c in citations.values() if c == canonical)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`'-> PASS' if overall >= 70 else '-> NEEDS WORK')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Local and Map SEO becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `consistent = sum(1 for c in citations.values() if c == canonical)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run a Full Local-SEO Health Report: connect the surface task to the chapter skill, check nap consistency across citations, score local signals, and audit business-profile completeness.
Names	Watch these names first: canonical, acme, plumbing, citations, Yelp. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: canonical = ('acme plumbing', '4155550199', '94110') | Core: consistent = sum(1 for c in citations.values() if c == canonical) | Evidence: '-> PASS' if overall >= 70 else '-> NEEDS WORK')

Reader to Writer Bridge

1. Read: explain run a full local-seo health report in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 431.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `'-> PASS' if overall >= 70 else '-> NEEDS WORK'`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `canonical = ('acme plumbing', '4155550199', '94110')` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run a full local-seo health report and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Combining the NAP consistency percentage with the profile completeness percentage into one average gives a headline health number that a threshold turns into a verdict.

```
canonical = ('acme plumbing', '4155550199', '94110')
citations = {
    'Yelp': ('acme plumbing', '4155550199', '94110'),
    'Bing': ('acme plumbing', '4155550100', '94110'),
    'Apple': ('acme plumbing', '4155550199', '94110'),
}
profile = {'name': 1, 'hours': 1, 'categories': 1, 'photos': 0, 'website': 1}
consistent = sum(1 for c in citations.values() if c == canonical)
nap_pct = 100 * consistent / len(citations)
prof_pct = 100 * sum(profile.values()) / len(profile)
overall = round((nap_pct + prof_pct) / 2, 1)
print(f'NAP consistency: {round(nap_pct, 1)}% ({consistent}/{len(citations)})')
print(f'profile complete: {round(prof_pct, 1)}%')
print('local SEO health:', overall, '%',
      '-> PASS' if overall >= 70 else '-> NEEDS WORK')
```

```
output
NAP consistency: 66.7% (2/3)
profile complete: 80.0%
local SEO health: 73.3 % -> PASS
```

How it works:

1. canonical and citations drive the NAP check, counting two of three sources matching for nap_pct.
2. profile maps five fields to present flags, and prof_pct sums them over the count for 80 percent.
3. overall averages nap_pct and prof_pct to 73.3 percent.
4. The print appends PASS because overall clears the seventy-percent threshold, else NEEDS WORK.

Try it:

1. Fold in review and proximity scores so the health metric covers more signals.
2. Make the pass threshold configurable rather than fixed at seventy.
3. Print the individual subscores so the verdict is explainable.
4. Fix the Bing phone so NAP consistency hits 100%.

⚠ Common Mistakes

- Averaging two percentages weights them equally; a real audit might weight NAP higher.
- Comparing the tuple citations with `==` requires identical normalization upstream first.
- Rounding the overall before the threshold check could nudge a 69.95 over the line; round only for display.

Skills gained health report | metric combination | pass-fail verdict

→ Where You'll Use This: A combined health score is the deliverable a local-SEO audit hands over: one number plus a verdict that tells a client at a glance whether their local presence is sound or needs work.

⚡ Mental Model

Like a doctors single fit-or-unfit summary that quietly rests on blood pressure, weight, and a dozen other readings underneath.

✅ Chapter Progress Checkpoint

What this chapter added: after 30 mini-projects, Local and Map SEO should feel like a reusable Python move: check nap consistency across citations, score local signals, and audit business-profile completeness.

chapter checkpoint

1 Read
Open any project from Projects 403-432 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 13 (Log-File and Server Analysis) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

7 questions · 4 categories · graded

2 Recall · 2 Meaning · 2 Structure · 1 Transfer

12.1.Q1 Recall · MEDIUM

Turn back to Project 419 (Detect a Profile Closed All Week). Without running it, what exactly does it print?

12.1.Q2 Meaning · MEDIUM

Turn back to Project 403 (Normalize a Business Name for Matching). Without running it, what exactly does it print?

12.1.Q3 Structure · MEDIUM

Turn back to Project 405 (Normalize a Street Address). Without running it, what exactly does it print?

12.1.Q4 Transfer · EASY

What are Google's three stated local-ranking factors?

- A. Relevance, distance, prominence
- B. Title, meta, headings
- C. Crawl, index, rank
- D. Speed, links, content

12.1.Q5 Recall · MEDIUM

Two citations list a phone as "(415) 555-0199" and "+1 415-555-0199". After normalizing to bare digits and dropping a leading US country code, are they consistent?

- A. No, the formatting differs
- B. Yes — both reduce to 4155550199
- C. Only if the area codes are removed
- D. Cannot tell without the address

12.1.Q6 Meaning · MEDIUM

You audit five citations and three exactly match your canonical NAP. What is your NAP consistency, and why does it matter?

12.1.Q7 Structure MEDIUM Why is adding "Plumber" into your Google Business Profile *name* field a bad idea, even though it seems relevant?

- A It costs extra
- B It violates Google's naming rules and risks suspension
- C It slows the page down
- D It has no effect at all

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 12 – Local and Map SEO

12.1.Q1. `WARNING: business marked closed all week` then `open day count: 0`

12.1.Q2. `"Joe's Coffee House, LLC"` then `normalized: "joe's coffee house"`

12.1.Q3. `raw: 123 North Main Street, Suite 200` then `norm: 123 n main st ste 200`

12.1.Q4. A relevance (does it match), distance (how near), prominence (how trusted/known).

12.1.Q5. B strip non-digits and drop the leading "1"; both become 4155550199, so they agree.

12.1.Q6. 60% (3 of 5). It matters because conflicting citations make Google less confident the listings describe one real business, which suppresses local ranking; fixing the two offenders is the highest-leverage move.

12.1.Q7. B the name field must be the real business name; keyword-stuffing it is against the rules and can get the profile suspended.

Chapter 13 · Log-File and Server Analysis

Parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.

Chapter Skill Map

What this chapter adds: In Log-File and Server Analysis, the focus is parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths. You practise it across 32 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples.

chapter arc

What it builds on: Chapter 13 assumes the reading habits from Chapter 12: Local and Map SEO. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 14, GEO, AEO and the End-to-End Technical Audit, reuses this chapter's discipline with a new kind of problem.

1 **Projects 433-442**	2 **Projects 443-452**	3 **Projects 453-464**
Recognise the pattern and read the code slowly. *name it first*	Trace similar programs with less prompting and explain what changed.	Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Your access log is the only record of what bots actually did on your site.

not what they should do — what they did

Every other SEO tool tells you what *should* happen: the sitemap lists the URLs you want crawled, robots.txt lists what you want blocked. The access log is different — it is the server's own diary of every request that arrived, written one line per hit, in the order they happened. It is ground truth. If Googlebot fetched a page you deleted, the log says so. If it never touched a page in your sitemap, the log says that too.

Most servers write the Common Log Format (CLF), a single space-separated line that packs the client IP, a timestamp, the request line, the response status, the bytes sent, and the User-Agent into one string. Learn to unpack that one line and you can read the whole file. A single regular expression with capture groups pulls every field out at once:

```
import re
line = ('66.249.66.1 - - [22/Jun/2026:10:00:00 +0000] '
        '"GET /page HTTP/1.1" 200 1234 "-" '
        '"Mozilla/5.0 (compatible; Googlebot/2.1)"')
m = re.match(
    r'(\S+) \S+ \S+ \[([^\]]+)\] "(\S+) (\S+) (\S+)" '
    r'(\d+) (\d+) "([^"]*)" "([^"]*)"', line)
ip, ts, method, path, proto, status, nbytes, ref, ua = m.groups()
print('ip:', ip)
print('path:', path)
```

```
    print('status:', status)
    print('ua:', ua)
```

```
output
ip: 66.249.66.1
path: /page
status: 200
ua: Mozilla/5.0 (compatible; Googlebot/2.1)
```

Read those nine groups and you have read a log line in full. The two fields that matter most for SEO are the `path` (which URL was asked for) and the `status` (how the server answered). The User-Agent tells you *who* asked, the timestamp tells you *when*, and the byte count tells you *how much* you served. Everything in this chapter is built from those few fields.

Why does any of this matter? Because a bot has a finite `crawl budget` — only so many fetches per visit. Spend them on pages that return 404, on redirect chains, or on a thousand parameter variants of one page, and your real pages wait days to be re-crawled. The log is where you *measure* that waste, count it, and find exactly which URLs are leaking budget.

you cannot fix what you have not counted

ACCESS LOG

The server-written record of every request received, one line per hit, in arrival order. It is the ground truth of what bots and users actually fetched.

COMMON LOG FORMAT (CLF)

The standard log line layout: `IP - - [timestamp] "METHOD path PROTOCOL" status bytes "referer" "user-agent"`. Each field is space-separated; quoted fields can contain spaces.

REQUEST PATH

The URL that was asked for, taken from the request line. The single most useful field for SEO log analysis because it names the page.

STATUS CODE

The three-digit response code (200, 301, 404, 503). Grouped by hundreds into classes: 2xx success, 3xx redirect, 4xx client error, 5xx server error.

USER-AGENT

The self-reported client string. Search bots announce themselves in it (`Googlebot`, `bingbot`), which is how you split bot traffic from human traffic.

CRAWL WASTE

Bot fetches spent on URLs that return nothing useful — a 404, a redirect, a blocked path, or a near-duplicate parameter URL. Every wasted fetch is budget stolen from a page that matters.

One line, nine fields

A CLF line looks like noise until you see its shape. The fields are positional: IP, two placeholders (`- -`), a bracketed timestamp, then the quoted request line holding method, path and protocol, then status, bytes, referer and User-Agent. You rarely need all nine. Most reports want just the `path` or just the `status`, so a focused regex that grabs only that token is faster to write and harder to break than a full parse. Here is the path, pulled from a mix of request lines:

```
import re
lines = [
    '1.2.3.4 - - [...] "GET /home HTTP/1.1" 200 512 "-" "Firefox"',
    '1.2.3.5 - - [...] "POST /api/login HTTP/1.1" 200 88 "-" "curl"',
    '1.2.3.6 - - [...] "GET /blog/post?id=7 HTTP/1.1" 200 999 "-" "Safari"',
]
for ln in lines:
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    print(path)
```

```
output
/home
/api/login
/blog/post?id=7
```

The pattern `"\S+ (\S+) HTTP` reads as: a quote, the method (non-spaces), a space, then the *captured* path (non-spaces), a space, then the protocol token. Because the path is captured, `.group(1)` returns just it. The status code lives a few fields later, and it arrives as text — parse it to `int` so you can do numeric range checks like `300 <= code < 400`:

```
import re
for raw in ['"GET /home HTTP/1.1" 200', '"GET /old HTTP/1.1" 301',
            '"GET /gone HTTP/1.1" 404']:
    status = int(re.search(r'(\d{3})$', raw).group(1))
    print(status, type(status).__name__)
```

```
output
200 int
301 int
404 int
```

Telling a real bot from a human

Search engines identify themselves in the User-Agent string. Googlebot carries `Googlebot` in it, Bing carries `bingbot`, Ahrefs carries `AhrefsBot`. So the first split — the one every later report depends on — is bot versus human, done by testing for marker substrings. Lowercase both sides so case never trips you up. A reliable way to grab the User-Agent is to split on the last two quote pairs, since it is the final quoted field:

```
lines = [
    '1.2.3.4 - - [...] "GET /a HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
Googlebot/2.1)"',
    '1.2.3.5 - - [...] "GET /b HTTP/1.1" 200 1 "-" "Mozilla/5.0 (Windows NT 10.0)
Firefox/120"',
    '1.2.3.6 - - [...] "GET /c HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
bingbot/2.0)"',
]
bots = ('googlebot', 'bingbot')
for ln in lines:
    ua = ln.rsplit('"', 2)[1]
    is_bot = any(b in ua.lower() for b in bots)
    print('BOT ' if is_bot else 'HUMAN', ua)
```

```
output
BOT   Mozilla/5.0 (compatible; Googlebot/2.1)
HUMAN Mozilla/5.0 (Windows NT 10.0) Firefox/120
BOT   Mozilla/5.0 (compatible; bingbot/2.0)
```

`ln.rsplit('"', 2)` splits from the right at most twice, so `[1]` is the text between the last two quotes — the User-Agent — no matter how many quotes appeared earlier. Once you can label each line, the headline number is the *share* of traffic that is crawlers versus people. That one ratio frames every crawl-budget question that follows.

know the bot share before you tune anything

Counting: paths, statuses, agents

A log is thousands of repeated paths and statuses, so the workhorse is the tally. A `Counter` over a parsed field gives raw demand for each value in one pass. Count paths to see what bots and users actually want; `most_common(n)` then surfaces the top hitters. Crawl budget concentrates on a few URLs, and this is how you see which:

```
import re
from collections import Counter
lines = [
    '"GET /home HTTP/1.1" 200', '"GET /blog HTTP/1.1" 200',
    '"GET /home HTTP/1.1" 200', '"GET /shop HTTP/1.1" 200',
```

```
    '"GET /home HTTP/1.1" 200', '"GET /blog HTTP/1.1" 200',
]
paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)
for path, n in paths.most_common(2):
    print(f'{path}: {n}')
```

```
output
/home: 3
/blog: 2
```

The same move counts status codes. Individual codes are noisy, so a cleaner health check groups them into classes by integer-dividing the code by 100 — `code // 100` turns 200, 301, 404 and 503 into 2, 3, 4 and 5. A spike in the 4xx or 5xx bucket jumps out instantly:

```
import re
from collections import Counter
raws = ['" 200 1 ', '" 301 0 ', '" 404 9 ', '" 200 1 ', '" 503 9 ']
classes = Counter()
for raw in raws:
    code = int(re.search(r'(\d{3})', raw).group(1))
    classes[f'{code // 100}xx'] += 1
for cls in sorted(classes):
    print(f'{cls}: {classes[cls]}')
```

```
output
2xx: 2
3xx: 1
4xx: 1
5xx: 1
```

Measuring crawl waste and budget

Now combine the splits. Crawl waste is budget a bot spent on URLs that returned nothing useful — most simply, a bot that fetched a 404 or chased a 3xx redirect. Filter for *bot* lines whose status is 404 or in the 3xx range, count them, and you have quantified the leak. This is the single most actionable number a log produces:

```
import re
lines = [
    '"GET /a HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '"GET /b HTTP/1.1" 404 9 "-" "Googlebot/2.1"',
    '"GET /c HTTP/1.1" 301 0 "-" "Googlebot/2.1"',
    '"GET /d HTTP/1.1" 200 1 "-" "Firefox/120"',
    '"GET /e HTTP/1.1" 404 9 "-" "bingbot/2.0"',
]
bots = ('googlebot', 'bingbot')
bot_hits = waste = 0
for ln in lines:
```

```
    ua = ln.rsplit('"', 2)[1].lower()
    code = int(re.search(r'" (\d{3}) ', ln).group(1))
    if any(b in ua for b in bots):
        bot_hits += 1
        if code == 404 or 300 <= code < 400:
            waste += 1
print('bot hits:', bot_hits)
print('wasted:', waste, '(', round(100 * waste / bot_hits, 1), '%)')
```

```
output
bot hits: 4
wasted: 3 ( 60.0 % )
```

Sixty percent of this bot's fetches hit dead ends — a budget emergency. Two more leaks worth counting: bots crawling paths your robots file disallows (test `path.startswith(d)` for each disallowed prefix), and bots crawling many URL-parameter variants of one page (count hits where `'?' in path`). Both are budget you did not want spent. And to size the budget itself, count the distinct paths a bot reached — a `set` of parsed paths collapses repeats automatically:

```
import re
lines = [
    '"GET /home HTTP/1.1" 200', '"GET /blog HTTP/1.1" 200',
    '"GET /home HTTP/1.1" 200', '"GET /shop HTTP/1.1" 200',
    '"GET /blog HTTP/1.1" 200',
]
paths = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}
print('total hits:', len(lines))
print('unique paths crawled:', len(paths))
```

```
output
total hits: 5
unique paths crawled: 3
```

Five fetches, three distinct URLs — the bot re-visited pages rather than reaching new ones. Comparing the crawled set against your sitemap closes the loop: `crawled - sitemap` finds URLs the bot chased that you never listed (often junk), and `sitemap - crawled` finds listed pages the bot never reached (orphans in practice). The log turns crawl budget from a vague worry into a counted, fixable list.

Example 1

A bot made 5 fetches: /home 200, /gone 404, /old 301, /home 200, /missing 404. What is the crawl-waste percentage, and which fix matters most?

SOLUTION

1. Count the fetches that returned a dead end. A 404 or any 3xx counts as waste: `/gone (404)`, `/old (301)`, `/missing (404)` = 3 wasted.
2. Total bot fetches = 5, so waste = 3 / 5 = `60%` of this bot's budget hit nothing useful.
3. The two 404s are deleted pages the bot still has on file; the 301 is a redirect it should not need to chase. Fixing the dead links removes 2 of the 3 wasted fetches.

60% crawl waste. Prioritise the 404s — they are confirmed dead links search engines are still spending budget on. *a logged 404 is a precise repair ticket*

Example 2

You parse 6 lines: 4 carry "Googlebot" in the User-Agent, 2 carry "Firefox". Among the Googlebot lines, statuses are 200, 200, 404, 301. What two numbers do you report?

SOLUTION

1. First split bot from human by testing for the marker substring `googlebot:` `4 bot hits`, 2 human hits.
2. Within the 4 bot hits, classify each status: 200 and 200 are useful 2xx; 404 and 301 are waste (a 4xx and a 3xx).
3. Crawl waste = 2 wasted / 4 bot hits = `50%`. Report the bot share (4 of 6 ≈ 66.7%) and the waste rate (50%).

Bot share ≈ 66.7% of traffic; crawl waste = 50% of bot fetches. Half of Googlebot's visits here achieved nothing. *two ratios summarise a whole log*

Example 3

A bot fetched these paths: /p?id=1, /p?id=2, /p?id=3, /blog. Group by the path before the "?". Where is the crawl bloat?

SOLUTION

1. Strip each path at the first `?` to get its base: /p?id=1, /p?id=2, /p?id=3 all reduce to base `/p`; /blog has no query so it stays /blog.
2. Tally bases: /p appears 3 times, /blog appears once.
3. A base with many parameter variants is duplicate-ish content soaking up fetches — /p burned 3 fetches on near-duplicates while /blog took one.

/p is crawl bloat: 3 parameter variants of one page. The fix is to canonicalise or block the parameter so the bot stops re-fetching duplicates. *one page, many query strings = wasted budget*

⚡ Mental-Math Shortcut

The log is ground truth; count, then split, then divide. First *count* the raw hits per path and per status. Then *split* bot from human on the User-Agent. Then *divide* to get the two numbers that matter: bot share of traffic, and crawl waste as a percentage of bot fetches. A sitemap says what should happen; the log is the only thing that says what did.

⚠ Common Mistakes

- Trusting the User-Agent as proof of identity. The string is self-reported, so a scraper can claim to be Googlebot — verify a suspicious bot by reverse-DNS on its IP before treating its hits as real crawler traffic.
- Splitting a log line on every space. Quoted fields like the request line and the User-Agent contain spaces; use a CLF-aware regex or rsplit on the quotes, not a naive `split()`.
- Counting only 404s as crawl waste and ignoring 3xx. A bot chasing redirect chains burns budget too, and a redirect to another redirect can quietly eat several fetches per target.
- Treating raw hit counts as the whole story. Three fetches of one URL is not the same as three distinct URLs crawled — track unique paths to measure crawl breadth, not just volume.

This chapter contains 32 hands-on mini-projects (Projects 433–464). Type each one out, run it, and experiment.

Project 433 Parse One Log Line into Its Fields

💡 The Idea Behind It

A raw access-log line is one dense string, yet every field sits in a fixed position. This project builds a single Common Log Format regex with nine capture groups and unpacks IP, timestamp, method, path, protocol, status, bytes, referrer and user-agent in one match. Understanding that anatomy is the foundation for every later analysis in the chapter.

What You'll Build & Learn

Take one server log line and split it into named, labelled fields you can read.

- Common Log Format field order
- regex capture groups
- tuple unpacking from match.groups

🧭 Skill Roadmap

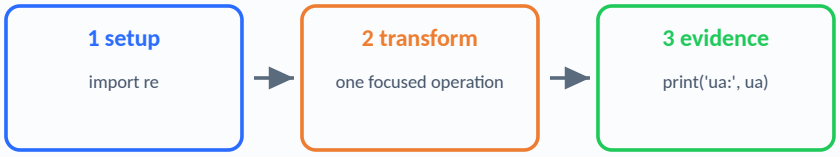
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 434 changes the surface problem while preserving the underlying method. *progress check*

Reading target: parse one log line into its fields is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The import tells you the project leans on a tool. Read the imported name as part of the design: the program is choosing a proven operation instead of rebuilding it.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('ua:', ua)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, transform applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`line = '66.249.66.1 - - [22/Jun/2026:10:00:00 +0000] "GET /page HTTP/1.1" 200 1234 "-" "Mozilla/5.0 (compatible; Googlebot/2.1)"'`	This line is the central rule of the tiny program. Slow down here and say what changes from before to after. *slow line*
3. Check the evidence	`print('ua:', ua)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE

```
line = '66.249.66.1 - - [22/Jun/2026:10:00:00 +0000] "GET /page HTTP/1.1" 200 1234 "-" "Mozilla/5.0 (compatible; Googlebot/2.1)"'
```

Construction pattern

The construction begins by importing tools. Read imported names as dependencies that the rest of the file relies on.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Underline the value being created and circle the operation that makes the final output possible.

Rewrite drill

Change one literal, name, or argument and explain which later line is affected first.

Code Reading Stack

Layer	What to read
Intent	Parse One Log Line into Its Fields: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, line, Jun, page, Mozilla. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: line = '66.249.66.1 - - [22/Jun/2026:10:00:00 +0000] "GET /page HTTP/1.... | Evidence: print('ua:', ua)

Reader to Writer Bridge

1. Read: explain parse one log line into its fields in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 434 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('ua:', ua)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of parse one log line into its fields and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

One `re.match` with parenthesised groups splits the whole line into named pieces; `\[([^\]]+)\]` grabs the bracketed timestamp and `"([^"]*)"` grabs each quoted field.

```
import re
line = '66.249.66.1 - - [22/Jun/2026:10:00:00 +0000] "GET /page HTTP/1.1" 200
1234 "-" "Mozilla/5.0 (compatible; Googlebot/2.1)"'
m = re.match(r'(\S+) \S+ \S+ \[([^\]]+)\] "(\S+) (\S+) (\S+)" (\d+) (\d+) "
([^"]*)" "([^"]*)"', line)
ip, ts, method, path, proto, status, nbytes, ref, ua = m.groups()
print('ip:', ip)
print('method:', method)
print('path:', path)
print('status:', status)
print('bytes:', nbytes)
print('ua:', ua)
```

```
output
ip: 66.249.66.1
method: GET
path: /page
status: 200
bytes: 1234
ua: Mozilla/5.0 (compatible; Googlebot/2.1)
```

How it works:

1. A sample Googlebot line is assigned to the variable line.
2. re.match runs a pattern with nine parenthesised groups against the start of that string.
3. m.groups() returns a tuple that is unpacked into ip, ts, method, path, proto, status, nbytes, ref and ua.
4. Six print statements label and show the most useful extracted fields.

Try it:

1. Add a print line that also displays the captured timestamp and referrer fields.
2. Replace re.match with re.search and test a line that has leading whitespace.
3. Wrap the parse in a try and print a warning when a malformed line fails to match.

4. Print `proto` to confirm the protocol token parsed as `HTTP/1.1`.

⚠ Common Mistakes

- Forgetting the brackets are literal in the timestamp group lets the regex skip into the wrong field.
- A missing space between the request and status groups makes the whole match fail and `m` is `None`.
- Calling `.group()` on a `None` match (a malformed line) raises AttributeError, so real parsers guard for it.

Skills gained regex | log parsing | string fields

→ Where You'll Use This: Every log-analysis tool and SEO crawl audit begins by turning unreadable log lines into structured fields you can count, filter and group.

⚡ Mental Model

The regex is a cookie cutter pressed onto a strip of dough: one stamp cleanly separates every field into its own labelled shape.

Project 434 Pull the Request Path From Each Line

💡 The Idea Behind It

Of all the log fields, the requested URL path matters most to an SEO, because it names the page a visitor or crawler actually wanted. This project loops over several lines and applies a narrow regex anchored on the quote and the HTTP token, capturing only the middle path segment from each request and printing it. It proves you can target one field without parsing the whole line.

What You'll Build & Learn

Extract only the request path from each log line and print the bare URLs.

- anchoring a regex on quotes
- single-group extraction
- iterating over log lines

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 433: parse one log line into its fields. Project 435 changes the surface problem while preserving the underlying method.

progress check

Reading target: pull the request path from each line is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(path)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`for ln in lines:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(path)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for ln in lines:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Pull the Request Path From Each Line: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, Jun, home, Firefox. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: for ln in lines: | Evidence: print(path)

🛠️ Reader to Writer Bridge

1. Read: explain pull the request path from each line in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 433.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 435 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(path)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of pull the request path from each line and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`'"\S+ (\S+) HTTP` anchors on the quoted request, skips the method with `\S+`, and captures the path before the protocol token.

```
import re
lines = [
    '1.2.3.4 - - [22/Jun/2026:10:00:00 +0000] "GET /home HTTP/1.1" 200 512 "-"
"Firefox"',
    '1.2.3.5 - - [22/Jun/2026:10:00:01 +0000] "POST /api/login HTTP/1.1" 200 88
"-" "curl"',
    '1.2.3.6 - - [22/Jun/2026:10:00:02 +0000] "GET /blog/post?id=7 HTTP/1.1"
200 999 "-" "Safari"',
]
for ln in lines:
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    print(path)
```

```
output
/home
/api/login
/blog/post?id=7
```

How it works:

1. A list named lines holds three request lines using GET and POST.
2. The for loop runs re.search with a pattern that matches the quote, the method token, then captures the path before HTTP.
3. group(1) pulls just the captured path out of the match.
4. Each path, including one with a query string, is printed on its own line.

Try it:

1. Add a line containing a PUT request and confirm the path still extracts correctly.
2. Strip any query string by splitting each printed path on the question mark character.
3. Collect the paths into a list instead of printing, then print the list length at the end.
4. Add a line with a query string and confirm the `?id=...` survives in the path.

⚠ Common Mistakes

- Splitting on spaces instead of regex breaks on query strings that contain encoded spaces.
- Capturing `\S+` greedily without the trailing `HTTP` anchor can swallow the protocol token too.
- A request line missing the protocol (a malformed log) makes the search return `None`.

Skills gained `regex` `path extraction` `loops`

→ **Where You'll Use This:** Pulling the clean path is the first step before tallying which URLs get hit, comparing them to a sitemap, or spotting crawl bloat.

⚡ Mental Model

Think of each log line as a sentence and the path as the one noun you underline, ignoring all the surrounding grammar.

Project 435 Read the Status Code as an Integer

💡 The Idea Behind It

Status codes in a log are stored as plain text, but their meaning is numeric: ranges like 2xx, 3xx and 4xx drive how search engines treat a URL. This project pulls the three-digit code with a regex and wraps it in int, then prints the value alongside its type name to prove the conversion. Numeric codes unlock range comparisons that string codes cannot support.

What You'll Build & Learn

Parse the HTTP status field from each line and convert it to a real integer.

- matching three digits with d{3}
- int conversion from text
- checking a value's type name

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 434: pull the request path from each line. Project 436 changes the surface problem while preserving the underlying method. *progress check*

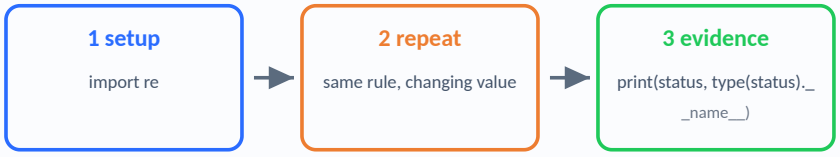
Reading target: read the status code as an integer is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.

- Evidence: `print(status, type(status).__name__)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`for ln in lines:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(status, type(status).__name__)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for ln in lines:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Read the Status Code as an Integer: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, Jun, home, Firefox. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: for ln in lines: | Evidence: print(status, type(status).__name__)

🛠️ Reader to Writer Bridge

1. Read: explain read the status code as an integer in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 434.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 436 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(status, type(status).__name__)`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of read the status code as an integer and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`" (\d{3}) ` matches exactly three digits after the quoted request, and `int(...)` converts the captured text so comparisons are numeric, not lexical.

```
import re
lines = [
    '1.2.3.4 - - [22/Jun/2026:10:00:00 +0000] "GET /home HTTP/1.1" 200 512 "-"
"Firefox"',
    '1.2.3.5 - - [22/Jun/2026:10:00:01 +0000] "GET /old HTTP/1.1" 301 0 "-"
"Firefox"',
    '1.2.3.6 - - [22/Jun/2026:10:00:02 +0000] "GET /gone HTTP/1.1" 404 153 "-"
"Safari"',
]
for ln in lines:
    status = int(re.search(r'" (\d{3}) ', ln).group(1))
    print(status, type(status).__name__)
```

```
output
200 int
301 int
404 int
```

How it works:

1. Three log lines carry the codes 200, 301 and 404.
2. Inside the loop re.search finds the quoted code surrounded by spaces and group(1) returns it as text.
3. int() converts that text into a number stored in status.
4. Each line prints the integer and type(status).__name__, confirming the value is now an int.

Try it:

1. Add a test that prints whether each code falls in the redirect range 300 to 399.
2. Append a 500 server-error line and confirm it parses and prints as an int.
3. Sum the parsed codes into a total just to see arithmetic now works on them.
4. Add a 502 line and confirm it parses as the integer 502.

⚠ Common Mistakes

- Leaving the code as a string makes `code < 400` compare characters, not numbers.
- Matching `\d+` instead of `\d{3}` could grab the byte count on an oddly spaced line.
- Forgetting the surrounding spaces lets the regex match three digits inside a longer number.

Skills gained regex | type conversion | status codes

→ Where You'll Use This: Numeric status codes let you write range tests for redirects and errors, the basis for crawl-health histograms and error reports later in this chapter.

⚡ Mental Model

Reading the code as text is like seeing a price tag as a sticker; calling int turns it into money you can actually add up.

Project 436 Flag Bot Hits by User-Agent

💡 The Idea Behind It

Search engines identify themselves inside the User-Agent string, so the first move in separating crawlers from people is spotting marker words. This project grabs the user-agent with rsplit on the quote character, lowercases it, and uses any() against a tuple of bot markers to label each hit BOT or HUMAN. It shows pattern matching on substrings rather than exact equality.

What You'll Build & Learn

Read each line's user-agent and label it as a bot or a human visitor.

- rsplit to grab the last quoted field
- case-insensitive substring matching
- any over a marker tuple

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 435: read the status code as an integer. Project 437 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag bot hits by user-agent is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

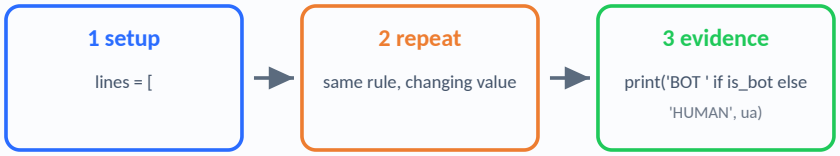
- Setup: `lines = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('BOT ' if is_bot else 'HUMAN', ua)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`lines = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`print('BOT ' if is_bot else 'HUMAN', ua)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('BOT ' if is_bot else 'HUMAN', ua)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('BOT ' if is_bot else 'HUMAN', ua)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Bot Hits by User-Agent: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: lines, Mozilla, compatible, Googlebot, Windows. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: lines = [| Core: print('BOT ' if is_bot else 'HUMAN', ua) | Evidence: print('BOT ' if is_bot else 'HUMAN', ua)

🛠️ Reader to Writer Bridge

1. Read: explain flag bot hits by user-agent in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 435.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 437 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('BOT ' if is_bot else 'HUMAN', ua)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `lines = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag bot hits by user-agent and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The user-agent is the last quoted field, so `rsplit('"', 2)[1]` lifts it out; lowercasing then testing membership against a marker tuple classifies the hit.

```
lines = [
    '1.2.3.4 - - [...] "GET /a HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
Googlebot/2.1)"',
    '1.2.3.5 - - [...] "GET /b HTTP/1.1" 200 1 "-" "Mozilla/5.0 (Windows NT
10.0) Firefox/120"',
    '1.2.3.6 - - [...] "GET /c HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
bingbot/2.0)"',
```

```
    '1.2.3.7 - - [...] "GET /d HTTP/1.1" 200 1 "-" "Mozilla/5.0 (Macintosh)
Safari/605"',
]
bots = ('googlebot', 'bingbot')
for ln in lines:
    ua = ln.rsplit('"', 2)[1]
    is_bot = any(b in ua.lower() for b in bots)
    print('BOT ' if is_bot else 'HUMAN', ua)
```

```
output
BOT   Mozilla/5.0 (compatible; Googlebot/2.1)
HUMAN Mozilla/5.0 (Windows NT 10.0) Firefox/120
BOT   Mozilla/5.0 (compatible; bingbot/2.0)
HUMAN Mozilla/5.0 (Macintosh) Safari/605
```

How it works:

1. A list of four lines mixes Googlebot, bingbot, Firefox and Safari agents.
2. bots holds the lowercase markers googlebot and bingbot.
3. Each loop pass uses ln.rsplit on the quote, index 1, to isolate the user-agent string.
4. any() tests whether a marker appears in the lowercased agent, and the result prints as BOT or HUMAN beside the agent.

Try it:

1. Add AhrefsBot and DuckDuckBot to the markers tuple and retest the lines.
2. Print a running count of how many lines were tagged as bots.
3. Handle a line with no quotes by guarding the rsplit with a length check.
4. Add an `AhrefsBot` line and extend the markers tuple to catch it.

⚠ Common Mistakes

- Matching case-sensitively misses `Googlebot` if you only test `googlebot` without lowercasing.
- Using `split('"')` from the left grabs the wrong quoted field when a referrer is present.
- A browser whose product token contains 'bot' would be a false positive worth noting.

Skills gained substring match | user-agent | rsplit

→ Where You'll Use This: Bot detection underpins crawl-budget analysis: nearly every later report splits traffic by whether a known search engine made the request.

Mental Model

It is like reading name tags at a conference door: a glance at the badge tells you whether someone is staff or a guest.

Project 437 Split Bot and Human Traffic

The Idea Behind It

Once a single hit can be classified, the natural summary is the overall split between crawler and human traffic. This project counts bot lines with a generator inside sum, where any() scans each lowercased line for a marker, then derives the human total by subtraction. The two printed numbers frame how much of your server's work is spent serving search engines versus actual people.

What You'll Build & Learn

Count how many log hits came from bots versus humans and print both totals.

- summing a boolean generator
- deriving the complement by subtraction
- marker tuples

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 436: flag bot hits by user-agent. Project 438 changes the surface problem while preserving the underlying method.

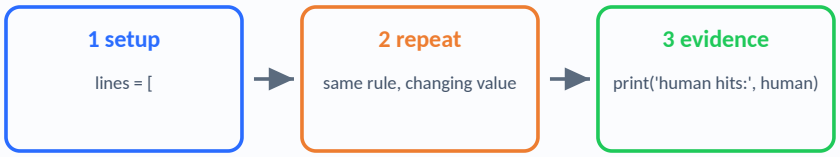
progress check

Reading target: split bot and human traffic is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `lines = [` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('human hits:', human)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`lines = [`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`bot = sum(1 for ln in lines if any(b in ln.lower() for b in bots))`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('human hits:', human)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `bot = sum(1 for ln in lines if any(b in ln.lower() for b in bots))`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Split Bot and Human Traffic: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: lines, Googlebot, Firefox, bingbot, Safari. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: lines = [| Core: bot = sum(1 for ln in lines if any(b in ln.lower() for b in bots)) | Evidence: print('human hits:', human)

🛠️ Reader to Writer Bridge

1. Read: explain split bot and human traffic in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 436.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 438 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('human hits:', human)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `lines = [` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split bot and human traffic and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Summing a generator of bot matches counts bot hits in one pass, and the human count is just the remainder of the total.

```
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 200 1 "-" "Firefox/120"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 200 1 "-" "bingbot/2.0"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 1 "-" "Safari/605"',
    '1.1.1.5 - - [...] "GET /e HTTP/1.1" 200 1 "-" "Chrome/119"',
]
```

```
bots = ('googlebot', 'bingbot')
bot = sum(1 for ln in lines if any(b in ln.lower() for b in bots))
human = len(lines) - bot
print('bot hits:', bot)
print('human hits:', human)
```

```
output
bot hits: 2
human hits: 3
```

How it works:

1. Five lines mix two bot agents with three human browser agents.
2. sum with a generator adds 1 for every line where any() finds a bot marker, giving bot.
3. human is computed as len(lines) minus bot rather than scanning again.
4. Both bot hits and human hits are printed.

Try it:

1. Print the bot share as a percentage of total hits using rounding.
2. Scan the full line versus only the user-agent field and compare the counts.
3. Add an unknown-agent line and decide which bucket it falls into.
4. Print the bot share as `round(100 * bot / len(lines), 1)` percent.

⚠ Common Mistakes

- Testing the marker against the whole line risks matching a path that happens to contain 'bot'.
- Recomputing `len(lines)` after filtering would double-count if you reuse the variable.
- Counting humans by re-scanning instead of subtracting can drift if the two passes disagree.

Skills gained counting generators traffic split

→ Where You'll Use This: The bot-versus-human ratio is a top-line metric in any log audit; a surprisingly high bot share is often the first sign of crawl-budget trouble.

⚡ Mental Model

Like a turnstile counter at a museum that separates ticketed visitors from staff, giving you the day's split at a glance.

Project 438 Count Requests per Path

💡 The Idea Behind It

To learn what users and crawlers actually want, you tally requests per URL. This project feeds the path extracted from each line into a Counter, which builds a tidy map of path to hit count in one expression. Printing the counts in sorted path order gives the raw demand profile for the site, the basis for understanding where attention concentrates.

What You'll Build & Learn

Count how many times each URL path appears across the log and print the tally.

- Counter from a generator
- regex path extraction
- sorted iteration of keys

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 437: split bot and human traffic. Project 439 changes the surface problem while preserving the underlying method.

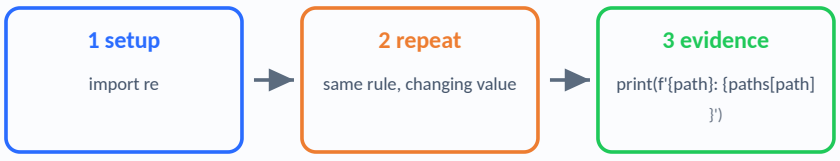
progress check

Reading target: count requests per path is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{path}: {paths[path]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {paths[path]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Requests per Path: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, home. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.

Layer	What to read
Evidence	Entry: import re | Core: paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in li... | Evidence: print(f'{path}: {paths[path]}')

🛠️ Reader to Writer Bridge

1. Read: explain count requests per path in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 437.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 439 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {paths[path]}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count requests per path and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Feeding a generator of parsed paths straight into `Counter` tallies hits per URL; sorting the keys before printing keeps the report deterministic.

```
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.2 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.3 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.4 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.5 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "Bot"',
]
paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)
for path in sorted(paths):
    print(f'{path}: {paths[path]}')
```

```
output
```

```
/blog: 2
/home: 3
```

How it works:

1. Five lines repeat the paths /home and /blog in varying amounts.
2. A Counter is built directly from a generator that runs re.search on each line and pulls group(1).
3. Counter records how many times each distinct path occurred.
4. The loop walks sorted(paths) and prints each path with its count using an f-string.

Try it:

1. Switch the output to print paths ordered by descending count instead of alphabetically.
2. Add several more lines and confirm the totals still add up correctly.
3. Print the single most-hit path along with its count.
4. Add another `/blog` hit and watch its count rise.

⚠ Common Mistakes

- Printing the Counter directly emits insertion order, which is not sorted or stable across runs.
- Counting full request lines instead of just the path treats identical paths as distinct.
- Forgetting to sort the keys makes the output order depend on hit order.

Skills gained Counter regex aggregation

→ **Where You'll Use This:** Per-URL hit counts feed crawl-frequency reports, top-page rankings and the parameter-bloat checks that appear later in the chapter.

⚡ Mental Model

The Counter is a tally clerk with a clipboard, adding a tick mark beside a URL every time someone asks for it.

Project 439 Surface the Most-Crawled Paths

💡 The Idea Behind It

Crawl budget concentrates on a handful of URLs, so ranking pages by hits answers where bots and users spend the most fetches. This project reuses the Counter-over-paths idea but calls most_common(2) to surface only the top two paths in descending order. It demonstrates that Counter does the sorting for you when you want a leaderboard rather than a full table.

What You'll Build & Learn

Rank URL paths by hit count and print only the two most-crawled pages.

- Counter.most_common with a limit

- ranking by frequency
- tuple unpacking in a loop

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 438: count requests per path. Project 440 changes the surface problem while preserving the underlying method.

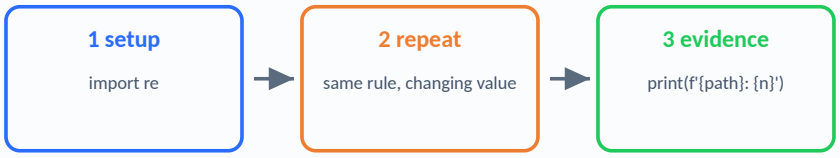
progress check

Reading target: surface the most-crawled paths is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{path}: {n}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
	`in lines)`	
3. Check the evidence	`print(f'{path}: {n}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Surface the Most-Crawled Paths: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, home. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in li... | Evidence: print(f'{path}: {n}')

Reader to Writer Bridge

1. Read: explain surface the most-crawled paths in one sentence before touching the code; the goal gives the syntax context.

2. Trace: follow the actual values from setup to output, then carry forward the move from Project 438.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 440 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {n}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of surface the most-crawled paths and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`Counter.most_common(N)` returns the top-N paths already sorted by descending count, so the busiest URLs surface without a manual sort.

```
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.2 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.3 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.4 - - [...] "GET /shop HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.5 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Bot"',
    '1.1.1.6 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "Bot"',
]
paths = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)
for path, n in paths.most_common(2):
    print(f'{path}: {n}')
```

```
output
/home: 3
/blog: 2
```

How it works:

1. Six lines give /home three hits, /blog two and /shop one.
2. A Counter tallies the path extracted from each line.
3. most_common(2) returns the top two path-count pairs already sorted high to low.
4. The loop unpacks each pair into path and n and prints them.

Try it:

1. Change the argument to most_common(3) and observe the extra row appear.
2. Call most_common with no argument to print every path in ranked order.
3. Add a tie between two paths and note how the order resolves.
4. Raise the top-N to 3 and see the full ranking.

⚠ Common Mistakes

- `most_common` breaks ties by insertion order, so equal counts are not alphabetical.
- Passing no argument to `most_common()` returns everything, not the top N.
- Counting request lines instead of paths splits one URL across many counts.

Skills gained most_common ranking Counter

→ Where You'll Use This: A top-pages leaderboard tells you which URLs deserve the freshest content and the tightest internal linking to keep crawlers returning.

⚡ Mental Model

It is a music chart: out of all the tracks played, you only print this week's top two singles.

Project 440 Tally Status Codes Across Log Lines

💡 The Idea Behind It

A frequency table of status codes is the quickest pulse check on a log, because a spike of 404s or 5xx leaps off the page. This project parses each three-digit code to an int and counts the values with a Counter, then prints them in sorted numeric order. The result is a compact histogram that summarises the health of every response the server returned.

What You'll Build & Learn

Build a frequency table of HTTP status codes across the log and print it sorted.

- counting integer keys
- regex plus int conversion
- sorted numeric output

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 439: surface the most-crawled paths. Project 441 changes the surface problem while preserving the underlying method.

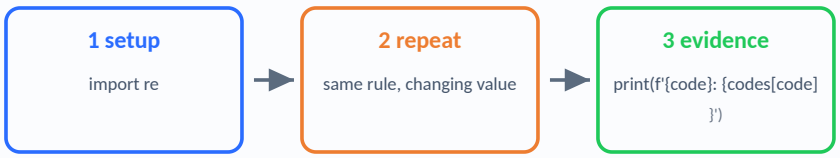
progress check

Reading target: tally status codes across log lines is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{code}: {codes[code]}')` exposes the result; match it against the output before changing the program. output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`codes = Counter(int(re.search(r'" (\d{3}) ', ln).group(1)) for ln in lines)`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{code}: {codes[code]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `codes = Counter(int(re.search(r'" (\d{3}) ', ln).group(1)) for ln in lines)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Tally Status Codes Across Log Lines: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, codes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: codes = Counter(int(re.search(r'" (\d{3}) ', ln).group(1)) for ln in li... | Evidence: print(f'{code}: {codes[code]}')

Reader to Writer Bridge

1. Read: explain tally status codes across log lines in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 439.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 441 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code}: {codes[code]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of tally status codes across log lines and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Parsing each code to `int` and feeding it to a `Counter` builds the histogram; sorting the codes prints them in numeric order.

```
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 301 0 "-" "B"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 404 9 "-" "B"',
    '1.1.1.5 - - [...] "GET /e HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.6 - - [...] "GET /f HTTP/1.1" 500 9 "-" "B"',
]
codes = Counter(int(re.search(r'" (\d{3}) ', ln).group(1)) for ln in lines)
for code in sorted(codes):
    print(f'{code}: {codes[code]}')
```

```
output
200: 3
301: 1
404: 1
500: 1
```

How it works:

1. Six lines carry the codes 200, 301, 404 and 500.
2. A Counter is built from a generator that converts each matched three-digit code with int.
3. Counter records how often each numeric code appeared.
4. Iterating sorted(codes) prints each code beside its count.

Try it:

1. Add several 404 lines and watch that bucket grow in the output.
2. Print only codes whose count exceeds one to highlight the common ones.
3. Compute the percentage each code represents of all hits.

4. Add a bar of `'#' * codes[code]` to draw the histogram.

⚠ Common Mistakes

- Counting codes as strings sorts '404' before '50' lexically, scrambling the histogram.
- Matching `\d+` could capture the byte field on a malformed line.
- Iterating `codes` unsorted gives a different order each run.

Skills gained Counter | status codes | histogram

→ Where You'll Use This: Status-code histograms are the standard at-a-glance log health check, flagging error surges that hurt crawlability before users complain.

Mental Model

Like a hospital chart of vital signs: one quick scan of the numbers tells you whether the server is healthy or feverish.

Project 441 Total the Bytes Served

The Idea Behind It

Bandwidth is a genuine cost and crawlers can consume a surprising slice of it, so totalling bytes served matters. This project extracts the byte field with a regex that skips past the status code, converts each value to int, and sums them inside a generator. It then prints the raw total and a kilobyte figure, showing how to aggregate a numeric column straight from text.

What You'll Build & Learn

Sum the bytes-served field across the log and report the total transfer.

- regex that skips the status to reach bytes
- summing converted integers
- rounding to kilobytes

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 440: tally status codes across log lines. Project 442 changes the surface problem while preserving the underlying method.

progress check

Reading target: total the bytes served is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('total KB:', round(total / 1024, 2))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`total = sum(int(re.search(r'" \d{3} (\d+) ', ln).group(1)) for ln in lines)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('total KB:', round(total / 1024, 2))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `total = sum(int(re.search(r'" \d{3} (\d+) ', ln).group(1)) for ln in lines)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Total the Bytes Served: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, total, sum, search. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: total = sum(int(re.search(r'" \d{3} (\d+) ', ln).group(1)) for ln in li... | Evidence: print('total KB:', round(total / 1024, 2))

🛠️ Reader to Writer Bridge

1. Read: explain total the bytes served in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 440.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 442 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('total KB:', round(total / 1024, 2))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of total the bytes served and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`" \d{3} (\d+) ` captures the byte count after the status code; summing the parsed integers totals bytes, and dividing by 1024 reports KB.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1200 "-" "B"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 200 3400 "-" "B"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 304 0 "-" "B"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 560 "-" "B"',
]
total = sum(int(re.search(r'" \d{3} (\d+) ', ln).group(1)) for ln in lines)
print('total bytes served:', total)
print('total KB:', round(total / 1024, 2))
```

```
output
total bytes served: 5160
total KB: 5.04
```

How it works:

1. Four lines carry byte sizes including a 304 response with zero bytes.
2. sum runs a generator that matches the status then captures the byte count and applies int.
3. The accumulated total is printed as raw bytes.
4. Dividing by 1024 and rounding to two places prints the size in KB.

Try it:

1. Add a megabyte line such as 5000000 bytes and reprint the KB total.
2. Also print the total in megabytes rounded to two decimals.
3. Skip lines whose byte field is zero before summing and compare totals.
4. Report the average bytes per request with `total / len(lines)`.

⚠ Common Mistakes

- A '-' byte field (some servers log it for 304s) would break `int(...)`; this fixture uses 0 instead.
- Capturing the status code by mistake inflates the total with response codes.
- Printing the raw float for KB without `round` gives a noisy long decimal.

Skills gained summing regex bandwidth

→ Where You'll Use This: Total transfer figures support bandwidth budgeting and later let you compare how much data bots pull versus humans on a busy site.

Mental Model

It is reading the odometer of data leaving your server: every response adds a few more miles to the trip total.

Project 442 Rank the Top User-Agents

The Idea Behind It

Knowing which clients hit a site most reveals whether crawlers, scrapers or real browsers dominate. This project counts the exact user-agent string, grabbed with rsplit on the quote, using a Counter, then prints every agent ranked by frequency via most_common. It treats the whole agent string as the key, so identical agents group together while distinct ones stay separate.

What You'll Build & Learn

Rank the user-agent strings in the log by how often each one appears.

- Counter over full user-agent strings
- rsplit field extraction
- most_common ranking

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 441: total the bytes served. Project 443 changes the surface problem while preserving the underlying method.

progress check

Reading target: rank the top user-agents is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.

- Evidence: `print(f'{n} {ua}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`agents = Counter(ln.rsplit('"', 2)[1] for ln in lines)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{n} {ua}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `agents = Counter(ln.rsplit('"', 2)[1] for ln in lines)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Rank the Top User-Agents: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: collections, Counter, lines, Googlebot, Firefox. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: agents = Counter(ln.rsplit('"', 2)[1] for ln in lines) | Evidence: print(f'{n} {ua}')

🛠️ Reader to Writer Bridge

1. Read: explain rank the top user-agents in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 441.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 443 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{n} {ua}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of rank the top user-agents and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The user-agent is the last quoted field via `rsplit('"', 2)[1]`; a `Counter` tallies them and `most_common()` prints the ranking.

```python
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 200 1 "-" "Firefox/120"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 1 "-" "bingbot/2.0"',
    '1.1.1.5 - - [...] "GET /e HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
]
agents = Counter(ln.rsplit('"', 2)[1] for ln in lines)
for ua, n in agents.most_common():
    print(f'{n}  {ua}')
```

```
output
3  Googlebot/2.1
1  Firefox/120
1  bingbot/2.0
```

How it works:

1. Five lines repeat Googlebot three times alongside one Firefox and one bingbot.
2. A Counter is built from each line's rsplit on the quote at index 1, the user-agent.
3. most_common with no argument returns every agent-count pair sorted by frequency.
4. The loop prints the count followed by the agent string.

Try it:

1. Normalise agents to a family name before counting so Googlebot versions merge.
2. Print only agents seen more than once.
3. Add an empty user-agent line and see how it ranks.
4. Limit the ranking with `most_common(2)`.

⚠ Common Mistakes

- Splitting from the left grabs the referrer field instead of the user-agent.
- Treating slightly different version strings as the same agent needs normalising first.
- `most_common()` ties break by first-seen order, not alphabetically.

Skills gained Counter | user-agent | ranking

→ Where You'll Use This: A top-agents table quickly exposes aggressive scrapers or unexpected bots that may be eating crawl budget or skewing your analytics.

⚡ Mental Model

Like a bouncer's headcount of who came through the door most often, by face rather than by category.

Project 443 Tally Requests by Status Class

💡 The Idea Behind It

Individual status codes are noisy, so grouping them into 2xx, 3xx, 4xx and 5xx classes gives a cleaner read on crawl health. This project parses each code, computes its class label with integer division by 100 plus the xx suffix, and increments a Counter under that label. Printing the classes in sorted order collapses dozens of codes into four meaningful buckets.

What You'll Build & Learn

Roll individual status codes up into their hundreds class and count each class.

- integer division to find a class
- f-string class labels
- Counter accumulation in a loop

🧭 Skill Roadmap

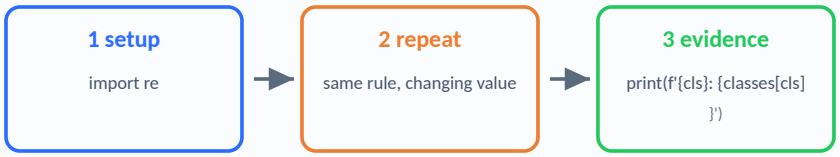
Skill step: moves you from recognition toward independent tracing. It builds on Project 442: rank the top user-agents. Project 444 changes the surface problem while preserving the underlying method. *progress check*

Reading target: tally requests by status class is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{cls}: {classes[cls]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`for ln in lines:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{cls}: {classes[cls]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for ln in lines:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Tally Requests by Status Class: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, classes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: for ln in lines: | Evidence: print(f'{cls}: {classes[cls]}')

🛠️ Reader to Writer Bridge

1. Read: explain tally requests by status class in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 442.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 444 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{cls}: {classes[cls]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of tally requests by status class and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Integer-dividing the code by 100 yields its class digit, and an f-string like `f'{code // 100}xx'` labels each bucket for the Counter.

```
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 301 0 "-" "B"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 404 9 "-" "B"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.5 - - [...] "GET /e HTTP/1.1" 503 9 "-" "B"',
```

```
]
classes = Counter()
for ln in lines:
    code = int(re.search(r'" (\d{3}) ', ln).group(1))
    classes[f'{code // 100}xx'] += 1
for cls in sorted(classes):
    print(f'{cls}: {classes[cls]}')
```

```
output
2xx: 2
3xx: 1
4xx: 1
5xx: 1
```

How it works:

1. Five lines span 200, 301, 404 and 503 codes.
2. An empty Counter named classes is created up front.
3. For each line the code is parsed to int, then code // 100 plus 'xx' builds labels like 2xx.
4. The matching class label is incremented, and sorted classes print with their counts.

Try it:

1. Print each class as a percentage of all hits.
2. Add a 1xx informational line and confirm a new 1xx bucket appears.
3. Flag the run as unhealthy when the 5xx class exceeds a threshold.
4. Add a 503 line and watch the 5xx bucket grow.

⚠ Common Mistakes

- Using string slicing `str(code)[0]` instead of `// 100` breaks on any non-three-digit code.
- Forgetting the `xx` label makes the buckets look like raw codes.
- Not sorting the class keys reorders the buckets between runs.

Skills gained integer division | Counter | status classes

→ Where You'll Use This: Status-class summaries are how dashboards report site health at a glance, separating successful responses from redirects, client errors and server faults.

Mental Model

Like sorting mail into four pigeonholes by floor number instead of reading every individual apartment address.

Project 444 Measure Crawl Waste From Bot Errors

The Idea Behind It

Crawl waste is budget spent on URLs that return nothing useful, such as a bot fetching a 404 or chasing a redirect. This project isolates bot lines, parses each status, and counts a line as wasted when a crawler hit a 404 or any 3xx code. It prints both total bot hits and wasted hits, putting a hard number on a leak that quietly drains crawl budget.

What You'll Build & Learn

Count how many bot requests landed on 404s or redirects, quantifying wasted crawl.

- combining bot and status filters
- redirect range test 300 to 400
- counting under two conditions

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 443: tally requests by status class. Project 445 changes the surface problem while preserving the underlying method.

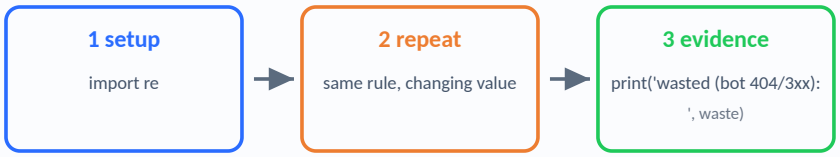
progress check

Reading target: measure crawl waste from bot errors is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('wasted (bot 404/3xx):', waste)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if any(b in ua for b in bots) and (code == 404 or 300 <= code < 400):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('wasted (bot 404/3xx):', waste)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if any(b in ua for b in bots) and (code == 404 or 300 <= code < 400):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure Crawl Waste From Bot Errors: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, Googlebot, Firefox, bingbot. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if any(b in ua for b in bots) and (code == 404 or 300 <= code < 400): | Evidence: print('wasted (bot 404/3xx):', waste)

🛠️ Reader to Writer Bridge

1. Read: explain measure crawl waste from bot errors in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 443.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 445 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('wasted (bot 404/3xx):', waste)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure crawl waste from bot errors and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

A hit counts as waste when the user-agent is a bot and the status is 404 or in the 3xx range, combining the bot test with a status-class check.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 404 9 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 301 0 "-" "Googlebot/2.1"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 1 "-" "Firefox/120"',
```

```
    '1.1.1.5 - - [...] "GET /e HTTP/1.1" 404 9 "-" "bingbot/2.0"',
]
bots = ('googlebot', 'bingbot')
waste = 0
for ln in lines:
    ua = ln.rsplit('"', 2)[1].lower()
    code = int(re.search(r'" (\d{3}) ', ln).group(1))
    if any(b in ua for b in bots) and (code == 404 or 300 <= code < 400):
        waste += 1
print('bot hits:', sum(1 for ln in lines if any(b in ln.lower() for b in
bots)))
print('wasted (bot 404/3xx):', waste)
```

```
output
bot hits: 4
wasted (bot 404/3xx): 3
```

How it works:

1. Five lines mix Googlebot, bingbot and Firefox hits with 200, 404 and 301 codes.
2. Each loop pass extracts the lowercased user-agent and parses the status to int.
3. A hit counts as waste when any() finds a bot marker and the code is 404 or sits in the 3xx range.
4. Total bot hits and the wasted count are both printed.

Try it:

1. Treat 5xx server errors as waste too and recount.
2. Print the waste figure as a percentage of bot hits.
3. Collect the wasted paths into a list so you know which URLs to fix.
4. Print the waste as a percentage of bot hits.

⚠ Common Mistakes

- Counting 3xx as waste is debatable: a single redirect is normal, chains are the real smell.
- Testing `code == 300` misses 301 and 302; the range `300 <= code < 400` is required.
- Including human 404s in the waste count answers a different question than crawl budget.

Skills gained filtering | crawl waste | conditions

→ Where You'll Use This: Quantifying wasted crawl makes the case for fixing dead links and redirect chains, the single biggest lever on how efficiently bots index your real pages.

Mental Model

It is like tracking how many delivery trips ended at a wrong or empty address: every one is fuel burned for no parcel delivered.

Project 445 Catch Bot Hits to Disallowed Paths

The Idea Behind It

When a bot crawls a path your robots file disallows, like /admin or /cart, either the rule is being ignored or the crawler is impolite, and either way it wastes budget. This project filters lines down to bot hits whose extracted path begins with a disallowed prefix, then prints each offending crawl. It combines user-agent detection with prefix matching against a disallow tuple.

What You'll Build & Learn

Flag bot requests that hit paths your robots rules are supposed to block.

- str.startswith for prefix rules
- combining bot and path filters
- tuple of disallowed prefixes

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 444: measure crawl waste from bot errors. Project 446 changes the surface problem while preserving the underlying method.

progress check

Reading target: catch bot hits to disallowed paths is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

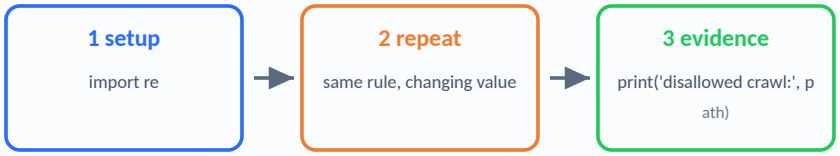
- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('disallowed crawl:', path)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if any(b in ua for b in bots) and any(path.startswith(d) for d in disallow):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('disallowed crawl:', path)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if any(b in ua for b in bots) and any(path.startswith(d) for d in disallow):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Catch Bot Hits to Disallowed Paths: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, home, Googlebot, admin. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if any(b in ua for b in bots) and any(path.startswith(d) for d in disal... | Evidence: print('disallowed crawl:', path)

🛠️ Reader to Writer Bridge

1. Read: explain catch bot hits to disallowed paths in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 444.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 446 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('disallowed crawl:', path)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of catch bot hits to disallowed paths and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For each bot hit, `path.startswith(prefix)` against a tuple of disallowed prefixes flags any crawl that should have been blocked.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /admin/x HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /cart HTTP/1.1" 200 1 "-" "bingbot/2.0"',
    '1.1.1.4 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
]
```

```
disallow = ('/admin', '/cart')
bots = ('googlebot', 'bingbot')
for ln in lines:
    ua = ln.rsplit('"', 2)[1].lower()
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    if any(b in ua for b in bots) and any(path.startswith(d) for d in
disallow):
        print('disallowed crawl:', path)
```

```
output
disallowed crawl: /admin/x
disallowed crawl: /cart
```

How it works:

1. Four lines include Googlebot and bingbot hits to /home, /admin/x, /cart and /blog.
2. disallow lists the prefixes /admin and /cart, and bots lists the crawler markers.
3. Each pass extracts the lowercased agent and the request path.
4. When any() confirms a bot and any path.startswith matches a disallowed prefix, the path prints as a disallowed crawl.

Try it:

1. Add /checkout to the disallowed prefixes and retest.
2. Print the offending bot's user-agent alongside each path.
3. Count the disallowed crawls instead of printing each one.
4. Add `/private` to the disallow tuple and a matching hit.

⚠ Common Mistakes

- Using `==` instead of `startswith` misses sub-paths like /admin/login under a /admin rule.
- Not lowercasing the user-agent misses `Googlebot` when matching against `googlebot`.
- A disallow prefix without a leading slash matches unintended substrings of the path.

Skills gained startswith robots rules filtering

→ Where You'll Use This: Catching bots inside disallowed areas validates your robots.txt and reveals sensitive sections leaking into crawl paths, a recurring technical-SEO audit task.

Mental Model

Like a security guard noticing a visitor wandering into rooms marked staff-only despite the sign on the door.

Project 446 List the 404s Googlebot Hit

The Idea Behind It

A 404 that Googlebot fetched is a known dead link search engines already have on file, so listing those paths yields a precise repair queue. This project filters lines to those where the user-agent contains googlebot and the parsed status equals 404, appending each path to a list. Sorting and printing the collected paths gives an actionable, deduplicatable record of broken URLs Google has seen.

What You'll Build & Learn

Collect and list every path where Googlebot received a 404 response.

- combining agent and exact-code filters
- appending matches to a list
- sorting collected paths

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 445: catch bot hits to disallowed paths. Project 447 changes the surface problem while preserving the underlying method.

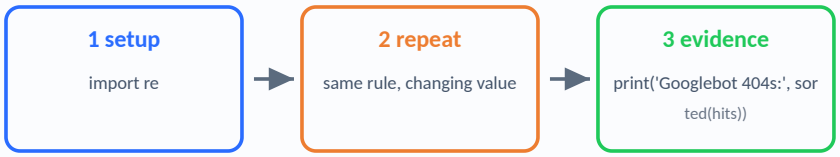
progress check

Reading target: list the 404s googlebot hit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('Googlebot 404s:', sorted(hits))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if 'googlebot' in ua and code == 404:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('Googlebot 404s:', sorted(hits))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if 'googlebot' in ua and code == 404:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	List the 404s Googlebot Hit: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, home, Googlebot, gone. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if 'googlebot' in ua and code == 404: | Evidence: print('Googlebot 404s:', sorted(hits))

🛠️ Reader to Writer Bridge

1. Read: explain list the 404s googlebot hit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 445.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 447 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('Googlebot 404s:', sorted(hits))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of list the 404s googlebot hit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Filtering hits where the user-agent contains 'googlebot' and the status equals 404 isolates the dead URLs Google still tries to crawl.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /gone HTTP/1.1" 404 9 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /missing HTTP/1.1" 404 9 "-" "bingbot/2.0"',
    '1.1.1.4 - - [...] "GET /old HTTP/1.1" 404 9 "-" "Googlebot/2.1"',
]
```

```
hits = []
for ln in lines:
    ua = ln.rsplit('"', 2)[1].lower()
    code = int(re.search(r'" (\d{3}) ', ln).group(1))
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    if 'googlebot' in ua and code == 404:
        hits.append(path)
print('Googlebot 404s:', sorted(hits))
```

```
output
Googlebot 404s: ['/gone', '/old']
```

How it works:

1. Four lines include Googlebot and bingbot hits with both 200 and 404 codes.
2. An empty list named hits is created.
3. Each pass extracts the lowercased agent, the integer status and the path.
4. When the agent contains googlebot and the code is exactly 404, the path is appended, and sorted hits prints at the end.

Try it:

1. Include bingbot 404s and label which crawler hit each path.
2. Deduplicate the list with a set before printing.
3. Also capture 410 Gone responses in the same pass.
4. Include bingbot by widening the user-agent test.

⚠ Common Mistakes

- Sorting the result is needed for stable output; a raw list follows log order.
- Matching status as a string `== '404'` works only if you never converted to int elsewhere.
- Counting every 404 ignores the question, which is specifically Googlebot's 404s.

Skills gained filtering lists 404 audit

→ **Where You'll Use This:** A Googlebot 404 list is a direct to-do for the SEO team: each path either needs a redirect, a restored page, or removal from internal links.

Mental Model

It is a list of addresses where the postman keeps finding the house demolished, handed straight to the maintenance crew.

Project 447 Count the Unique Paths Crawled

The Idea Behind It

The count of distinct URLs in a log measures crawl breadth, how widely bots and users roam the site. This project collects every parsed path into a set comprehension, which discards repeats automatically, then prints the set size and the sorted unique paths. It contrasts with earlier Counter work by caring about presence rather than frequency.

What You'll Build & Learn

Count the distinct URL paths that appear in the log and list them.

- set comprehension for uniqueness
- len of a set
- sorting a set for display

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 446: list the 404s googlebot hit. Project 448 changes the surface problem while preserving the underlying method.

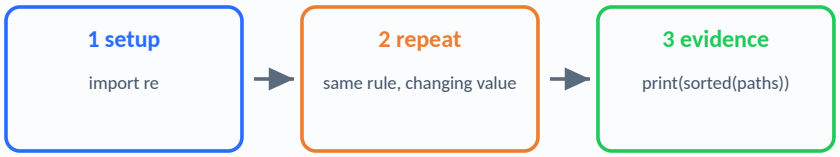
progress check

Reading target: count the unique paths crawled is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(sorted(paths))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`paths = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(sorted(paths))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `paths = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count the Unique Paths Crawled: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, home, blog, shop. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: paths = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines} | Evidence: print(sorted(paths))

🛠️ Reader to Writer Bridge

1. Read: explain count the unique paths crawled in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 446.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 448 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(sorted(paths))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count the unique paths crawled and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A set comprehension over the parsed paths drops duplicates, so its length is the count of unique URLs touched.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.3 - - [...] "GET /home HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.4 - - [...] "GET /shop HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.5 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "B"',
```

```
]
paths = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}
print('unique paths:', len(paths))
print(sorted(paths))
```

```
output
unique paths: 3
['/blog', '/home', '/shop']
```

How it works:

1. Five lines repeat /home, /blog and /shop with duplicates.
2. A set comprehension runs re.search on each line and keeps only group(1), automatically dropping repeats.
3. len(paths) gives the number of unique paths.
4. sorted(paths) prints the distinct URLs in order.

Try it:

1. Compare the unique count to the total line count to gauge repetition.
2. Strip query strings before adding paths so variants collapse.
3. Print which paths appear only once versus many times.
4. Add a duplicate `/home` hit and confirm the unique count does not change.

⚠ Common Mistakes

- Using a list keeps duplicates, so `len` overstates the unique count.
- Printing the set directly is nondeterministic; sort it first.
- Counting request lines instead of paths treats two methods on one URL as two URLs.

Skills gained sets uniqueness regex

→ Where You'll Use This: Crawl breadth tells you how much of a large site bots actually reach; a low unique-path count on a big site signals discoverability problems.

Mental Model

Like counting how many different rooms a tour group visited, no matter how many times anyone walked back through the same door.

Project 448 Report Crawl Frequency per URL

The Idea Behind It

Beyond raw counts, the share of total hits each URL receives shows where crawl attention truly concentrates. This project tallies paths with a Counter, sums the values to get a grand total, then prints each path's hit count

alongside its percentage of all hits. Converting counts to percentages turns a frequency table into a clear per-URL slice of the crawl pie.

What You'll Build & Learn

Report each URL's hit count and its percentage share of total crawl activity.

- summing Counter values for a total
- percentage of a part to a whole
- formatted percentage output

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 447: count the unique paths crawled. Project 449 changes the surface problem while preserving the underlying method.

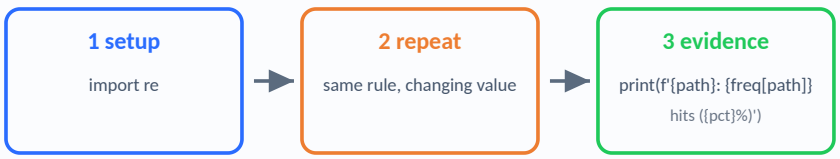
progress check

Reading target: report crawl frequency per url is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{path}: {freq[path]} hits ({pct}%)')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`freq = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {freq[path]} hits ({pct}%)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `freq = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Report Crawl Frequency per URL: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, home. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: freq = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lin... | Evidence: print(f'{path}: {freq[path]} hits ({pct}%)')

Reader to Writer Bridge

1. Read: explain report crawl frequency per url in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 447.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 449 should feel like a variation, not a brand-new mystery.

from example to tool

Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {freq[path]} hits ({pct}%)')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of report crawl frequency per url and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

From Concept to Code (Python)

Dividing each path's count by the summed total and multiplying by 100 gives its percentage share, rounded for clean output.

```python
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [...] "GET /home HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.3 - - [...] "GET /home HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.4 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.5 - - [...] "GET /shop HTTP/1.1" 200 1 "-" "B"',
]
freq = Counter(re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines)
total = sum(freq.values())
for path in sorted(freq):
    pct = round(100 * freq[path] / total, 1)
    print(f'{path}: {freq[path]} hits ({pct}%)')
```

```
output
/blog: 1 hits (20.0%)
/home: 3 hits (60.0%)
/shop: 1 hits (20.0%)
```

How it works:

1. Five lines give /home three hits and /blog and /shop one each.
2. A Counter named freq tallies the extracted paths.
3. sum(freq.values()) computes the grand total of hits.
4. For each sorted path, the share is computed as 100 times count over total, rounded, and printed with the count.

Try it:

1. Sort the output by descending percentage instead of by path.
2. Add a cumulative percentage column to see how few URLs cover most hits.
3. Round the percentages to two decimal places for precision.
4. Sort the report by share descending.

⚠ Common Mistakes

- Dividing by `len(freq)` (number of URLs) instead of total hits gives the wrong denominator.
- Integer division would floor every percentage to 0; the counts must be floats here.
- Not rounding prints long repeating decimals like 33.33333.

Skills gained `percentages` `Counter` `reporting`

→ **Where You'll Use This:** Percentage shares make crawl reports comparable across sites of different sizes and reveal when one page hogs a disproportionate slice of attention.

Mental Model

It turns a stack of tally marks into pie slices, so you can see at a glance which page eats the biggest wedge.

Project 449 Find Crawled URLs Off the Sitemap

The Idea Behind It

A bot crawling URLs absent from your sitemap may be chasing junk: faceted parameters, old links, or pages you forgot to list. This project builds a set of crawled paths and a set of sitemap URLs, then uses set subtraction to find paths that were crawled but never declared. The off-sitemap result points straight at URLs that may be wasting crawl or hint at gaps in your sitemap.

What You'll Build & Learn

Find crawled URLs that do not appear in your sitemap using set subtraction.

- set difference with the minus operator
- building a crawled-path set

- comparing log to sitemap

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 448: report crawl frequency per url. Project 450 changes the surface problem while preserving the underlying method.

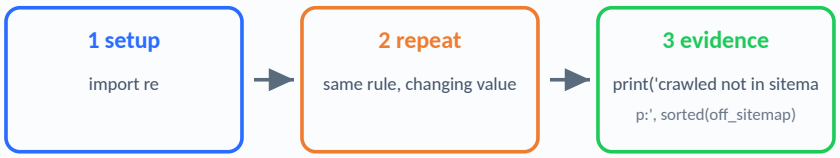
progress check

Reading target: find crawled urls off the sitemap is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('crawled not in sitemap:', sorted(off_sitemap))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print('crawled not in sitemap:', sorted(off_sitemap))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find Crawled URLs Off the Sitemap: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, home, Googlebot, ghost. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines} | Evidence: print('crawled not in sitemap:', sorted(off_sitemap))

Reader to Writer Bridge

1. Read: explain find crawled urls off the sitemap in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 448.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 450 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('crawled not in sitemap:', sorted(off_sitemap))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find crawled urls off the sitemap and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Subtracting the sitemap set from the set of crawled paths leaves exactly the URLs that were fetched but never declared in the sitemap.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /ghost HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "bingbot/2.0"',
    '1.1.1.4 - - [...] "GET /stale HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
]
sitemap = {'/home', '/blog', '/shop'}
crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}
off_sitemap = crawled - sitemap
print('crawled not in sitemap:', sorted(off_sitemap))
```

```
output
crawled not in sitemap: ['/ghost', '/stale']
```

How it works:

1. Four lines include Googlebot and bingbot hits to /home, /ghost, /blog and /stale.
2. sitemap is a literal set of declared URLs.
3. crawled is a set comprehension of every parsed path from the log.
4. crawled minus sitemap yields off_sitemap, which prints sorted as the URLs crawled but not listed.

Try it:

1. Swap the operands to instead list sitemap URLs never crawled.
2. Restrict the crawled set to only bot hits before comparing.

3. Print how many off-sitemap URLs were found, not just the list.
4. Add `/ghost` to the sitemap and watch it drop off the list.

⚠ Common Mistakes

- Subtracting in the wrong order returns sitemap-only URLs, a different report.
- Comparing lists instead of sets makes the `-` operator fail.
- Not sorting the difference makes the output order vary.

Skills gained set difference | sitemap | crawl audit

→ Where You'll Use This: Comparing crawl logs against the sitemap surfaces orphaned or stray URLs that dilute crawl budget and may not belong in the index at all.

Mental Model

It is checking the guest list against who actually walked in: the difference is the gatecrashers nobody invited.

Project 450 Break Down the HTTP Method Mix

The Idea Behind It

GET dominates crawl traffic, but a flood of POST or HEAD requests can signal forms being probed or uptime monitors at work. This project captures the HTTP method, the first token after the opening quote, with a tight regex and feeds it into a Counter. Printing the methods in sorted order reveals the request mix and any unusual verb activity worth investigating.

What You'll Build & Learn

Break down the log by HTTP method and count how often each verb is used.

- capturing the leading method token
- Counter over methods
- sorted method output

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 449: find crawled urls off the sitemap. Project 451 changes the surface problem while preserving the underlying method.

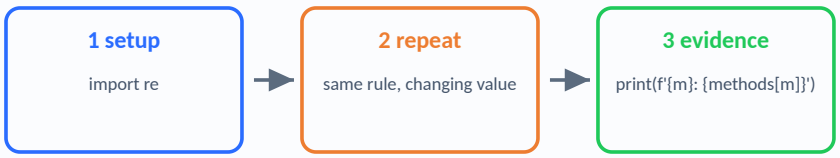
progress check

Reading target: break down the http method mix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{m}: {methods[m]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`methods = Counter(re.search(r'" (\S+) ', ln).group(1) for ln in lines)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{m}: {methods[m]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `methods = Counter(re.search(r'"(\S+) ', ln).group(1) for ln in lines)`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Break Down the HTTP Method Mix: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, home. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: methods = Counter(re.search(r'"(\S+) ', ln).group(1) for ln in lines) | Evidence: print(f'{m}: {methods[m]}')

🛠️ Reader to Writer Bridge

1. Read: explain break down the http method mix in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 449.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 451 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{m}: {methods[m]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of break down the http method mix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`'"(\S+) ` captures the leading method token inside the quoted request, and a `Counter` tallies how often each verb appears.

```
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [...] "POST /login HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.3 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.4 - - [...] "HEAD /ping HTTP/1.1" 200 0 "-" "B"',
    '1.1.1.5 - - [...] "GET /shop HTTP/1.1" 200 1 "-" "B"',
]
methods = Counter(re.search(r'"(\S+) ', ln).group(1) for ln in lines)
for m in sorted(methods):
    print(f'{m}: {methods[m]}')
```

```
output
GET: 3
HEAD: 1
POST: 1
```

How it works:

1. Five lines use GET three times plus one POST and one HEAD.
2. A Counter is built from a generator that matches the opening quote and captures the method token.
3. Counter tallies each distinct verb.
4. Iterating sorted(methods) prints each method with its count.

Try it:

1. Flag a warning when POST requests exceed a chosen share of traffic.
2. Add a DELETE line and confirm it gets its own bucket.
3. Print methods ranked by count rather than alphabetically.
4. Add a DELETE line and watch a new bucket appear.

⚠ Common Mistakes

- Capturing the path by mistake counts URLs, not methods.
- Lowercase methods would split GET from get; logs normally keep them uppercase.
- Not sorting the method keys reorders the breakdown between runs.

Skills gained `regex` `Counter` `HTTP methods`

→ **Where You'll Use This:** Method breakdowns help spot scraping, form abuse or monitoring traffic that inflates request volume without representing genuine page demand.

⚡ Mental Model

Like sorting incoming requests by the kind of action they ask for, GET to read, POST to submit, HEAD to peek at the headers.

Project 451 Detect a Crawl Spike by the Minute

💡 The Idea Behind It

A sudden burst of requests in one minute can mean a crawler went rogue or a page got hammered. This project parses each timestamp with strptime, buckets hits by the HH:MM minute via strftime, and counts them in a Counter. Each minute prints with a SPIKE flag when its count exceeds a threshold, turning raw timestamps into a simple anomaly detector.

What You'll Build & Learn

Bucket log hits by the minute and flag any minute that exceeds a request threshold.

- strptime to parse log timestamps
- bucketing by minute with strftime
- threshold flagging

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 450: break down the http method mix. Project 452 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect a crawl spike by the minute is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from datetime import datetime` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.

- Evidence: `print(f'{minute}: {per_min[minute]}{flag}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from datetime import datetime`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`flag = ' SPIKE' if per_min[minute] > threshold else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{minute}: {per_min[minute]}{flag}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `flag = ' SPIKE' if per_min[minute] > threshold else ''`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Crawl Spike by the Minute: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: datetime, collections, Counter, lines, Jun. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from datetime import datetime | Core: flag = ' SPIKE' if per_min[minute] > threshold else '' | Evidence: print(f'{minute}: {per_min[minute]}{flag}')

🛠️ Reader to Writer Bridge

1. Read: explain detect a crawl spike by the minute in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 450.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 452 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{minute}: {per_min[minute]}{flag}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `from datetime import datetime` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a crawl spike by the minute and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`datetime.strptime` parses the bracketed timestamp, `strftime('%H:%M')` buckets each hit by minute, and a count over the threshold is flagged a spike.

```
from datetime import datetime
from collections import Counter
lines = [
    '1.1.1.1 - - [22/Jun/2026:10:00:01 +0000] "GET /a HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [22/Jun/2026:10:00:30 +0000] "GET /b HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.3 - - [22/Jun/2026:10:00:45 +0000] "GET /c HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.4 - - [22/Jun/2026:10:00:55 +0000] "GET /d HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.5 - - [22/Jun/2026:10:01:10 +0000] "GET /e HTTP/1.1" 200 1 "-" "B"',
]
per_min = Counter()
for ln in lines:
    raw = ln.split('[')[1].split()[0]
    t = datetime.strptime(raw, '%d/%b/%Y:%H:%M:%S')
    per_min[t.strftime('%H:%M')] += 1
threshold = 3
for minute in sorted(per_min):
    flag = ' SPIKE' if per_min[minute] > threshold else ''
    print(f'{minute}: {per_min[minute]}{flag}')
```

```
output
10:00: 4 SPIKE
10:01: 1
```

How it works:

1. Five lines fall mostly in 10:00 with one at 10:01.
2. The timestamp is sliced out between the bracket and the space, then parsed by strptime with the log format string.
3. strftime formats it to HH:MM and that minute key is incremented in per_min.
4. Each sorted minute prints its count, appending SPIKE when the count tops the threshold of three.

Try it:

1. Lower the threshold to two and watch more minutes flag.
2. Bucket by ten-second windows to catch sharper bursts.
3. Only count bot hits when deciding whether a minute spikes.
4. Lower the threshold to 2 and watch more buckets flag.

⚠ Common Mistakes

- Using `datetime.now` is forbidden and wrong here; the timestamps come from the log itself.
- Bucketing by `%H` (hour) instead of `%H:%M` hides minute-level bursts.
- Comparing `>=` instead of `>` flags a bucket exactly at the threshold.

Skills gained `datetime` `bucketing` `anomaly flag`

→ **Where You'll Use This:** Per-minute spike detection catches runaway crawlers and traffic floods early, before they exhaust crawl budget or knock the server over.

⚡ Mental Model

Like a seismograph for traffic: most minutes draw a flat line, and a spike is the tremor that makes you look.

Project 452 Count Requests per Bot Type

💡 The Idea Behind It

Different crawlers have different appetites, so attributing each hit to a named bot shows whether Googlebot, bingbot or a scraper does the work. This project loops over a tuple of known bot names and, for each line, increments a Counter under whichever name appears in the lowercased user-agent. Printing the per-bot counts gives a clear breakdown of who is crawling and how hard.

What You'll Build & Learn

Attribute each bot hit to a specific named crawler and count them per bot.

- matching against a list of bot names
- nested loop over known bots
- Counter keyed by bot name

🧭 Skill Roadmap

`Skill step:` moves you from recognition toward independent tracing. It builds on Project 451: detect a crawl spike by the minute. Project 453 changes the surface problem while preserving the underlying method.

progress check

`Reading target:` count requests per bot type is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `from collections import Counter` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` A nested loop means `outer choice` plus `inner repeated work`. Read one full inner pass before jumping ahead; that reveals the grid, table, or pairwise pattern.
- `Writing move:` this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.

- Evidence: `print(f'{bot}: {counts[bot]}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from collections import Counter`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if bot.lower() in ua.lower():`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{bot}: {counts[bot]}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if bot.lower() in ua.lower():`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Semicolon: Ends many Java/C/JavaScript statements or CSS declarations; missing one can join two ideas accidentally.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Requests per Bot Type: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: collections, Counter, lines, Mozilla, compatible. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from collections import Counter | Core: if bot.lower() in ua.lower(): | Evidence: print(f'{bot}: {counts[bot]}')

🛠️ Reader to Writer Bridge

1. Read: explain count requests per bot type in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 451.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 453 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{bot}: {counts[bot]}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from collections import Counter` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count requests per bot type and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For each line the user-agent is tested against a tuple of known bot names, and the first matching name increments its Counter bucket.

```
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
Googlebot/2.1)"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
bingbot/2.0)"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
Googlebot/2.1)"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
AhrefsBot/7.0)"',
    '1.1.1.5 - - [...] "GET /e HTTP/1.1" 200 1 "-" "Mozilla/5.0 (compatible;
Googlebot/2.1)"',
]
known = ('Googlebot', 'bingbot', 'AhrefsBot')
counts = Counter()
for ln in lines:
    ua = ln.rsplit('"', 2)[1]
    for bot in known:
        if bot.lower() in ua.lower():
            counts[bot] += 1
for bot in sorted(counts):
    print(f'{bot}: {counts[bot]}')
```

```
output
AhrefsBot: 1
Googlebot: 3
bingbot: 1
```

How it works:

1. Five lines feature Googlebot, bingbot and AhrefsBot agents.
2. known holds the three bot names and an empty Counter is created.
3. For each line the user-agent is isolated, then an inner loop checks each known bot name in lowercase.
4. The matching bot name is incremented, and sorted counts print per bot.

Try it:

1. Add a bucket for unknown agents that match none of the known bots.
2. Extend the known list with YandexBot and DuckDuckBot.
3. Print each bot's share of total bot hits as a percentage.
4. Add a `YandexBot` line and extend the known tuple.

⚠ Common Mistakes

- A user-agent matching two names (rare) would be counted twice without a break.
- Case-sensitive matching misses `Googlebot` when the known list is lowercase.

- Not sorting the bot keys reorders the per-bot report.

Skills gained `Counter` `bot attribution` `nested loops`

→ Where You'll Use This: Naming each crawler tells you which search engines value your site and which third-party scrapers are quietly consuming resources you may want to throttle.

Mental Model

It is a visitor log at reception that records not just that a courier came, but which delivery company they worked for.

Project 453 Average Response Size by Status

The Idea Behind It

A 200 should carry a real payload while a 404 should be tiny, so averaging bytes per status code surfaces bloated error pages or empty success responses. This project captures both code and bytes in one regex, groups byte sizes into a defaultdict of lists keyed by status, then prints the mean size for each code. Grouping numeric values by a category and averaging is a core analysis pattern.

What You'll Build & Learn

Group response sizes by status code and report the average bytes for each code.

- defaultdict of lists for grouping
- capturing two fields in one regex
- computing a mean per group

Skill Roadmap

`Skill step:` moves you from recognition toward independent tracing. It builds on Project 452: count requests per bot type. Project 454 changes the surface problem while preserving the underlying method.

progress check

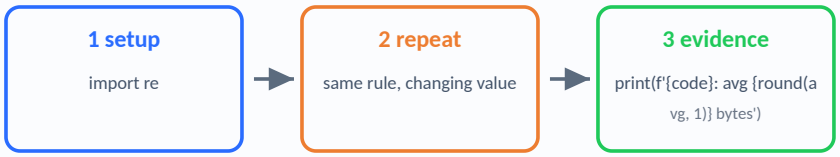
`Reading target:` average response size by status is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- `Setup:` `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- `Core logic:` The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as `data -> transformation -> evidence`, not as isolated library calls.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{code}: avg {round(avg, 1)} bytes')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`for ln in lines:`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print(f'{code}: avg {round(avg, 1)} bytes')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for ln in lines:`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Average Response Size by Status: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, defaultdict, lines, sizes. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: for ln in lines: | Evidence: print(f'{code}: avg {round(avg, 1)} bytes')

🛠️ Reader to Writer Bridge

1. Read: explain average response size by status in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 452.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 454 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code}: avg {round(avg, 1)} bytes')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of average response size by status and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Grouping byte counts into a `defaultdict(list)` keyed by status code lets you average each group, exposing how heavy each response class is.

```
import re
from collections import defaultdict
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1000 "-" "B"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 200 2000 "-" "B"',
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 404 150 "-" "B"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 3000 "-" "B"',
    '1.1.1.5 - - [...] "GET /e HTTP/1.1" 404 250 "-" "B"',
]
sizes = defaultdict(list)
for ln in lines:
    code, nbytes = re.search(r'" (\d{3}) (\d+) ', ln).groups()
    sizes[int(code)].append(int(nbytes))
for code in sorted(sizes):
    avg = sum(sizes[code]) / len(sizes[code])
    print(f'{code}: avg {round(avg, 1)} bytes')
```

```
output
200: avg 2000.0 bytes
404: avg 200.0 bytes
```

How it works:

1. Five lines pair 200 and 404 codes with various byte sizes.
2. A defaultdict(list) named sizes is created.
3. One regex captures the status and byte fields together, and each byte value is appended to the list under its integer code.
4. For each sorted code, the average of its list prints rounded as bytes.

Try it:

1. Also print the maximum byte size seen for each status code.
2. Add a 200 line with zero bytes and watch the average shift.
3. Group by status class such as 2xx instead of exact code.
4. Print the max byte size per status alongside the average.

⚠ Common Mistakes

- Dividing by `len(lines)` instead of the per-code count gives the wrong average.
- Capturing only the status and not the bytes leaves nothing to average.
- Printing the raw float without `round` yields noisy decimals.

Skills gained `defaultdict` `averaging` `grouping`

→ **Where You'll Use This:** Average size by status flags error pages that are surprisingly heavy or success pages returning almost nothing, both worth investigating for performance and SEO.

Mental Model

Like weighing every parcel and then reporting the average weight per shipping label, so an oddly heavy category stands out.

Project 454 Find the Busiest Hour

The Idea Behind It

Knowing your peak traffic hour helps schedule deploys away from it and spot crawler activity windows. This project parses each timestamp, buckets hits by the integer hour into a Counter, then picks the busiest hour with max using the count as the key. It prints that hour formatted with leading zeros and its hit total, demonstrating how to find an argmax over a frequency table.

What You'll Build & Learn

Find the hour of day with the most log hits and print it with its count.

- bucketing by the hour attribute
- max with a key function
- zero-padded hour formatting

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 453: average response size by status. Project 455 changes the surface problem while preserving the underlying method. *progress check*

Reading target: find the busiest hour is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

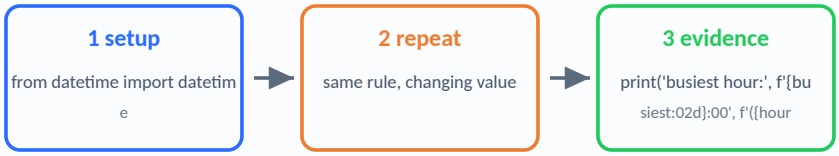
Logic Lens

- Setup: `from datetime import datetime` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('busiest hour:', f'{busiest:02d}:00', f'({hours[busiest]} hits)')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from datetime import datetime`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`for ln in lines:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('busiest hour:', f'{busiest:02d}:00', f'({hours[busiest]} hits)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for ln in lines:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find the Busiest Hour: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: datetime, collections, Counter, lines, Jun. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from datetime import datetime | Core: for ln in lines: | Evidence: print('busiest hour:', f'{busiest:02d}:00', f'({hours[busiest]} hits)')

🛠️ Reader to Writer Bridge

1. Read: explain find the busiest hour in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 453.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 455 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('busiest hour:', f'{busiest:02d}:00', f'({hours[busiest]} hits)')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `from datetime import datetime` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find the busiest hour and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

`strptime(...).hour` extracts the hour from each parsed timestamp; `max` over the sorted hours with a count key returns the busiest one deterministically.

```
from datetime import datetime
from collections import Counter
lines = [
    '1.1.1.1 - - [22/Jun/2026:09:14:00 +0000] "GET /a HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [22/Jun/2026:10:01:00 +0000] "GET /b HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.3 - - [22/Jun/2026:10:30:00 +0000] "GET /c HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.4 - - [22/Jun/2026:10:55:00 +0000] "GET /d HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.5 - - [22/Jun/2026:11:02:00 +0000] "GET /e HTTP/1.1" 200 1 "-" "B"',
]
hours = Counter()
for ln in lines:
    raw = ln.split('[')[1].split()[0]
    hours[datetime.strptime(raw, '%d/%b/%Y:%H:%M:%S').hour] += 1
busiest = max(sorted(hours), key=lambda h: hours[h])
print('busiest hour:', f'{busiest:02d}:00', f'({hours[busiest]} hits)')
```

```
output
busiest hour: 10:00 (3 hits)
```

How it works:

1. Five lines span the 09, 10 and 11 hours, with three in the 10 hour.
2. Each timestamp is sliced out and parsed by strptime, and its .hour attribute increments the hours Counter.
3. max over sorted(hours) with a key of hours[h] returns the hour with the highest count.
4. That busiest hour prints zero-padded as HH:00 with its hit total.

Try it:

1. Print the three busiest hours instead of only the top one.
2. Restrict the count to bot hits to find the peak crawl hour.
3. Bucket by day and hour together for a multi-day log.
4. Print every hour with its hit count, not just the peak.

⚠ Common Mistakes

- Calling `max` over the unsorted Counter makes ties resolve unpredictably.
- Taking `max` of the counts loses which hour they belong to.
- Parsing with the wrong format string raises ValueError on the bracketed timestamp.

Skills gained `datetime` `argmax` `Counter`

→ Where You'll Use This: Peak-hour analysis guides maintenance windows and reveals when search engines crawl most, helping you ensure servers respond fast during critical times.

⚡ Mental Model

Like asking a shop when its busiest hour is so you avoid restocking the shelves while customers are crowding the aisles.

Project 455 Track First and Last Seen per Path

💡 The Idea Behind It

When a URL first and last appeared in the log bounds its activity window, telling you whether a page is freshly crawled or has gone quiet. This project keeps a dict mapping each path to a (first, last) tuple, updating it per line with min and max against the parsed timestamp. Printing the window per path shows each URL's span of activity in a compact form.

What You'll Build & Learn

Track the first and last time each URL path was seen in the log.

- dict.get with a default tuple
- min and max to widen a window
- per-key state updates in a loop

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 454: find the busiest hour. Project 456 changes the surface problem while preserving the underlying method. *progress check*

Reading target: track first and last seen per path is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop is the engine: it repeats the same rule over changing values. Name the value that changes each pass, then the repeated action becomes easy to predict.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{path}: {f.strftime("%H:%M")} -> {l.strftime("%H:%M")}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`for ln in lines:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{path}: {f.strftime("%H:%M")} -> {l.strftime("%H:%M")}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `for ln in lines:`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Track First and Last Seen per Path: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, datetime, lines, Jun, home. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: for ln in lines: | Evidence: print(f'{path}: {f.strftime("%H:%M")} -> {l.strftime("%H:%M")}')

🛠️ Reader to Writer Bridge

1. Read: explain track first and last seen per path in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 454.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 456 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {f.strftime("%H:%M")} -> {l.strftime("%H:%M")}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of track first and last seen per path and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For each path the running min and max of its parsed timestamps are kept in a dict, so the final pair is the page's first and last appearance.

```
import re
from datetime import datetime
lines = [
    '1.1.1.1 - - [22/Jun/2026:10:00:00 +0000] "GET /home HTTP/1.1" 200 1 "-"
"B"',
    '1.1.1.2 - - [22/Jun/2026:10:05:00 +0000] "GET /blog HTTP/1.1" 200 1 "-"
"B"',
    '1.1.1.3 - - [22/Jun/2026:10:09:00 +0000] "GET /home HTTP/1.1" 200 1 "-"
"B"',
    '1.1.1.4 - - [22/Jun/2026:10:12:00 +0000] "GET /home HTTP/1.1" 200 1 "-"
"B"',
]
seen = {}
for ln in lines:
    raw = ln.split('[')[1].split()[0]
    t = datetime.strptime(raw, '%d/%b/%Y:%H:%M:%S')
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    first, last = seen.get(path, (t, t))
    seen[path] = (min(first, t), max(last, t))
for path in sorted(seen):
    f, l = seen[path]
    print(f'{path}: {f.strftime("%H:%M")} -> {l.strftime("%H:%M")}')
```

```
output
/blog: 10:05 -> 10:05
/home: 10:00 -> 10:12
```

How it works:

1. Four lines hit /home three times and /blog once at different minutes.
2. An empty dict named seen is created.
3. Each line parses its timestamp and path, reads the existing window with seen.get defaulting to (t, t), then stores the min and max against the new time.
4. For each sorted path, the window prints as a start to end time range.

Try it:

1. Also record how many times each path was seen inside its window.

2. Compute and print the duration of each window in minutes.
3. Flag paths whose last-seen time is far behind the others.
4. Print the active span in minutes for each path.

⚠ Common Mistakes

- Initialising the pair to `(t, t)` on first sight is essential; an empty default breaks `min`/`max`.
- Comparing timestamp strings instead of `datetime` objects sorts wrong across hour boundaries.
- Overwriting instead of taking min/max loses the earlier or later time.

Skills gained dict state | datetime | min max

→ Where You'll Use This: First and last seen windows reveal which pages crawlers have stopped revisiting, an early warning that content is going stale in the index.

Mental Model

Like a guestbook that records each room's first and last visitor of the day, bracketing when it was in use.

Project 456 Detect Orphan Sitemap URLs

The Idea Behind It

A sitemap URL that never appears in the log is one search engines may not have crawled yet, an orphan in practice. This project builds a set of crawled paths from the log and subtracts it from the sitemap set, leaving the URLs that were declared but never fetched. The result is the mirror image of the off-sitemap check, pointing at pages that need help getting discovered.

What You'll Build & Learn

Find sitemap URLs that never appear in the crawl log using set subtraction.

- sitemap minus crawled for orphans
- set comprehension from the log
- interpreting an empty intersection

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 455: track first and last seen per path. Project 457 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect orphan sitemap urls is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('sitemap URLs never crawled:', sorted(orphans))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('sitemap URLs never crawled:', sorted(orphans))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Orphan Sitemap URLs: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, home, Googlebot, blog. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines} | Evidence: print('sitemap URLs never crawled:', sorted(orphans))

🛠️ Reader to Writer Bridge

1. Read: explain detect orphan sitemap urls in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 455.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 457 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('sitemap URLs never crawled:', sorted(orphans))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect orphan sitemap urls and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Subtracting the set of crawled paths from the sitemap set leaves URLs that were declared but never fetched in this log window.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /home HTTP/1.1" 200 1 "-" "bingbot/2.0"',
]
sitemap = {'/home', '/blog', '/shop', '/about'}
crawled = {re.search(r'"\S+ (\S+) HTTP', ln).group(1) for ln in lines}
orphans = sitemap - crawled
print('sitemap URLs never crawled:', sorted(orphans))
```

```
output
sitemap URLs never crawled: ['/about', '/shop']
```

How it works:

1. Three lines crawl only /home and /blog.
2. sitemap is a literal set of four declared URLs.
3. crawled is a set comprehension of every parsed path.
4. sitemap minus crawled yields orphans, which prints sorted as the sitemap URLs never crawled.

Try it:

1. Count and print how many orphans were found.
2. Restrict the crawled set to Googlebot hits only.
3. Run both directions to print orphans and off-sitemap URLs together.
4. Add a `/shop` hit and watch it leave the orphan list.

⚠ Common Mistakes

- Subtracting the sitemap from the crawled set answers the opposite question.

- A URL crawled with a query string will not match its clean sitemap entry without normalising.
- Not sorting the gap set makes the output order vary.

Skills gained set difference | orphan detection | sitemap

→ Where You'll Use This: Orphan detection highlights pages search engines are ignoring, prompting fixes like better internal linking or resubmitting the sitemap to speed discovery.

⚡ Mental Model

It is checking which invited guests never showed up: the empty chairs are the pages bots have not reached.

Project 457 Pull Redirect Hits From the Log

💡 The Idea Behind It

Redirects in a log, the 301, 302 and 308 codes, show where URLs send visitors and bots elsewhere. This project parses each status and path, then filters to the 3xx class with a range test, printing the redirect code and the path it was served on. Listing every redirect hit makes redirect behaviour visible so you can audit chains and outdated rules.

What You'll Build & Learn

Filter the log to redirect responses and list each redirecting path with its code.

- redirect range test 300 to 400
- extracting code and path together
- filtering inside a loop

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 456: detect orphan sitemap urls. Project 458 changes the surface problem while preserving the underlying method.

progress check

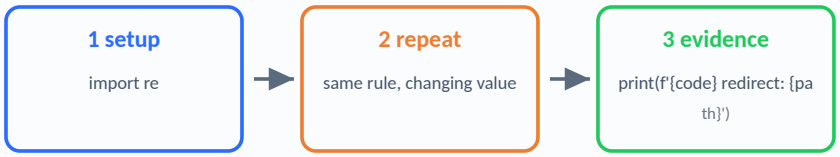
Reading target: pull redirect hits from the log is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{code} redirect: {path}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if 300 <= code < 400:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{code} redirect: {path}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if 300 <= code < 400:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Pull Redirect Hits From the Log: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, home, old, www. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if 300 <= code < 400: | Evidence: print(f'{code} redirect: {path}')

🛠️ Reader to Writer Bridge

1. Read: explain pull redirect hits from the log in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 456.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 458 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code} redirect: {path}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of pull redirect hits from the log and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Testing `300 <= code < 400` selects the redirect class, and the captured path names the URL that issued each redirect.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [...] "GET /old HTTP/1.1" 301 0 "-" "B"',
    '1.1.1.3 - - [...] "GET /www HTTP/1.1" 302 0 "-" "B"',
    '1.1.1.4 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.5 - - [...] "GET /moved HTTP/1.1" 308 0 "-" "B"',
]
for ln in lines:
    code = int(re.search(r'" (\d{3}) ', ln).group(1))
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    if 300 <= code < 400:
        print(f'{code} redirect: {path}')
```

```
output
301 redirect: /old
302 redirect: /www
308 redirect: /moved
```

How it works:

1. Five lines mix 200 responses with 301, 302 and 308 redirects.
2. Each pass parses the status to int and extracts the request path.
3. An if test keeps only codes where 300 is at most the code and the code is below 400.
4. Each qualifying line prints its redirect code and path.

Try it:

1. Separate permanent 301 and 308 from temporary 302 in the output.
2. Count how many redirects each path received.
3. Add a 307 line and confirm it is captured by the range.
4. Separate permanent (301/308) from temporary (302/307) redirects.

⚠ Common Mistakes

- Testing `code == 301` alone misses 302, 307 and 308 redirects.
- Treating 304 Not Modified as a redirect is wrong; it is a cache validator even though it is in the 3xx range.

- Matching status as a string skips the numeric range check entirely.

Skills gained `range filter` `redirects` `regex`

→ **Where You'll Use This:** A redirect inventory helps you find chains, loops and leftover temporary 302s that should be permanent, all of which dilute link signals and slow crawling.

Mental Model

Like reading the forwarding stickers on returned mail: each one tells you the old address and that the resident moved on.

Project 458 Spot Oversized Responses

The Idea Behind It

A multi-megabyte image or unminified asset wastes bandwidth and slows crawls, so flagging responses above a byte threshold finds the heavy hitters. This project parses the byte field and path from each line and prints any response larger than a set limit, converting the size to megabytes for readability. It is a simple but powerful filter for performance triage.

What You'll Build & Learn

Flag log responses whose size exceeds a byte limit and report them in megabytes.

- thresholding a numeric field
- bytes to megabytes conversion
- extracting bytes and path together

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 457: pull redirect hits from the log. Project 459 changes the surface problem while preserving the underlying method. *progress check*

Reading target: spot oversized responses is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

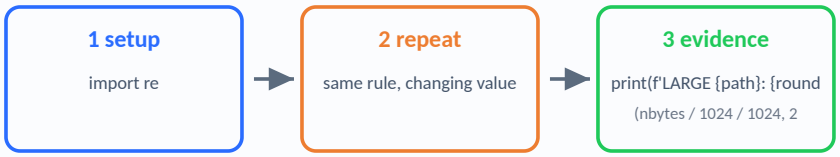
Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the

most-crawled paths.

- Evidence: `print(f'LARGE {path}: {round(nbytes / 1024 / 1024, 2)} MB')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if nbytes > limit:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'LARGE {path}: {round(nbytes / 1024 / 1024, 2)} MB')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if nbytes > limit:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Spot Oversized Responses: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, css, hero, jpg. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if nbytes > limit: | Evidence: print(f'LARGE {path}: {round(nbytes / 1024 / 1024, 2)} MB')

🛠️ Reader to Writer Bridge

1. Read: explain spot oversized responses in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 457.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 459 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'LARGE {path}: {round(nbytes / 1024 / 1024, 2)} MB')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of spot oversized responses and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Comparing the parsed byte count to a fixed limit flags large transfers, and dividing by 1024 twice reports the size in megabytes.

```python
import re
lines = [
    '1.1.1.1 - - [...] "GET /a.css HTTP/1.1" 200 4000 "-" "B"',
    '1.1.1.2 - - [...] "GET /hero.jpg HTTP/1.1" 200 920000 "-" "B"',
    '1.1.1.3 - - [...] "GET /b.js HTTP/1.1" 200 30000 "-" "B"',
    '1.1.1.4 - - [...] "GET /video.mp4 HTTP/1.1" 200 5500000 "-" "B"',
]
limit = 500000
for ln in lines:
    nbytes = int(re.search(r'" \d{3} (\d+) ', ln).group(1))
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    if nbytes > limit:
        print(f'LARGE {path}: {round(nbytes / 1024 / 1024, 2)} MB')
```

```
output
LARGE /hero.jpg: 0.88 MB
LARGE /video.mp4: 5.25 MB
```

How it works:

1. Four lines list a CSS file, a hero image, a JS file and a video with varying sizes.
2. limit is set to 500000 bytes.
3. Each pass parses the byte count and the path.
4. When the byte count exceeds the limit, the path prints with its size converted to megabytes and rounded.

Try it:

1. Sort the flagged responses from largest to smallest before printing.
2. Lower the limit to 100000 and see more assets flagged.
3. Group oversized hits by file extension to find the worst asset type.
4. Lower the limit to 100 KB and watch more assets flag.

⚠ Common Mistakes

- Using a threshold in bytes but reporting MB without converting confuses the reader.
- Capturing the status instead of the byte field compares the wrong number.
- An integer MB conversion would floor 0.88 MB to 0; keep it a float and round.

Skills gained thresholds | byte conversion | performance

→ Where You'll Use This: Spotting oversized assets drives image compression and minification work, directly improving page speed, a confirmed ranking and crawl-efficiency factor.

Mental Model

Like an airport scale beeping only when a bag tips over the weight limit, so you focus on the few that are too heavy.

Project 459 Compute the Error Rate per Path

The Idea Behind It

Some URLs fail more than others, so dividing error hits by total hits per path ranks pages by how often they break. This project keeps two defaultdicts, one counting all hits and one counting hits with a status of 400 or higher, per path. It then prints each path's error rate as a percentage with the supporting fraction, turning raw counts into a reliability score per URL.

What You'll Build & Learn

Compute and report the percentage of requests that errored for each URL path.

- two parallel defaultdicts
- error test code at least 400
- rate as errors over total

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 458: spot oversized responses. Project 460 changes the surface problem while preserving the underlying method.

progress check

Reading target: compute the error rate per path is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

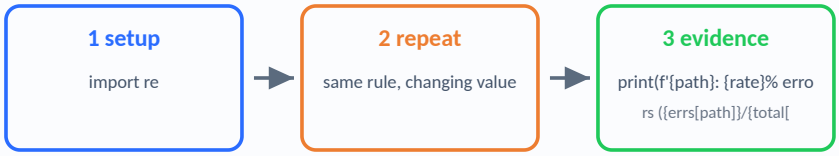
- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print(f'{path}: {rate}% errors ({errs[path]}/{total[path]})')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if code >= 400:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{path}: {rate}% errors ({errs[path]}/{total[path]})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if code >= 400:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Compute the Error Rate per Path: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, defaultdict, lines, total. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if code >= 400: | Evidence: print(f'{path}: {rate}% errors ({errs[path]}/{total[path]})')

🛠️ Reader to Writer Bridge

1. Read: explain compute the error rate per path in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 458.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 460 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {rate}% errors ({errs[path]}/{total[path]})')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of compute the error rate per path and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Two `defaultdict(int)` counters track total and error (status >= 400) hits per path, and their ratio is the per-URL error rate.

```
import re
from collections import defaultdict
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.2 - - [...] "GET /a HTTP/1.1" 500 9 "-" "B"',
    '1.1.1.3 - - [...] "GET /b HTTP/1.1" 200 1 "-" "B"',
    '1.1.1.4 - - [...] "GET /a HTTP/1.1" 200 1 "-" "B"',
```

```
    '1.1.1.5 - - [...] "GET /b HTTP/1.1" 404 9 "-" "B"',
]
total = defaultdict(int)
errs = defaultdict(int)
for ln in lines:
    code = int(re.search(r'" (\d{3}) ', ln).group(1))
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    total[path] += 1
    if code >= 400:
        errs[path] += 1
for path in sorted(total):
    rate = round(100 * errs[path] / total[path], 1)
    print(f'{path}: {rate}% errors ({errs[path]}/{total[path]})')
```

```
output
/a: 33.3% errors (1/3)
/b: 50.0% errors (1/2)
```

How it works:

1. Five lines hit /a and /b with a mix of 200, 404 and 500 codes.
2. total and errs are both defaultdict(int).
3. Each pass parses the status and path, always increments total, and increments errs only when the code is 400 or above.
4. For each sorted path the rate is errors over total times 100, printed with the underlying fraction.

Try it:

1. Sort the output by descending error rate to surface the worst pages.
2. Count only 5xx server errors rather than all 4xx and 5xx.
3. Skip paths with fewer than a minimum number of hits to reduce noise.
4. Flag any path over a 50% error rate.

⚠ Common Mistakes

- Counting 3xx redirects as errors inflates the rate; errors start at 400.
- Dividing by the global total instead of the per-path total mixes pages together.
- A path with zero hits would divide by zero, but every path here has at least one.

Skills gained defaultdict | error rate | percentages

→ Where You'll Use This: Per-path error rates pinpoint the specific URLs costing you crawl trust and user goodwill, prioritising which broken pages to repair first.

Mental Model

Like a batting average per player: it is not the raw misses that matter but how often each one fails out of all their attempts.

Project 460 Measure Crawl Budget Spent on Parameters

The Idea Behind It

Bots that crawl many URL-parameter variants of one page burn budget on near-duplicates, so measuring the parametered share quantifies the waste. This project counts bot hits whose path contains a question mark separately from clean bot hits, then prints both counts and the percentage of bot budget spent on parametered URLs. It frames query-string crawling as a measurable efficiency loss.

What You'll Build & Learn

Measure what share of a bot's requests target parametered URLs versus clean ones.

- detecting a query string with in
- splitting counts by a condition
- percentage of parametered hits

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 459: compute the error rate per path. Project 461 changes the surface problem while preserving the underlying method.

progress check

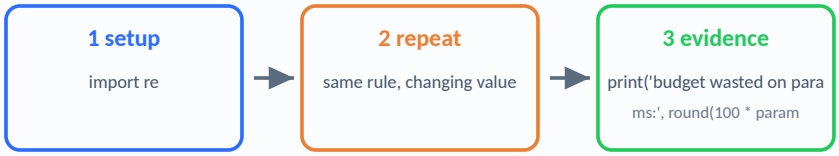
Reading target: measure crawl budget spent on parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('budget wasted on params:', round(100 * param / (param + clean), 1), '%')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if any(b in ua for b in bots):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('budget wasted on params:', round(100 * param / (param + clean), 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if any(b in ua for b in bots):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure Crawl Budget Spent on Parameters: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, utm, ad, Googlebot. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if any(b in ua for b in bots): | Evidence: print('budget wasted on params:', round(100 * param / (param + clean),...

🛠️ Reader to Writer Bridge

1. Read: explain measure crawl budget spent on parameters in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 459.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 461 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('budget wasted on params:', round(100 * param / (param + clean), 1), '%')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure crawl budget spent on parameters and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Among bot hits, a `?` in the path marks a parametered URL; the share of parametered hits estimates budget lost to query-string variants.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /p?utm=ad HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /p HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /p?sort=asc HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
```

```
    '1.1.1.4 - - [...] "GET /q HTTP/1.1" 200 1 "-" "bingbot/2.0"',
    '1.1.1.5 - - [...] "GET /p?utm=x&sort=z HTTP/1.1" 200 1 "-"
"Googlebot/2.1"',
]
bots = ('googlebot', 'bingbot')
param = clean = 0
for ln in lines:
    ua = ln.rsplit('"', 2)[1].lower()
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    if any(b in ua for b in bots):
        if '?' in path:
            param += 1
        else:
            clean += 1
print('bot hits with params:', param)
print('bot hits clean:', clean)
print('budget wasted on params:', round(100 * param / (param + clean), 1), '%')
```

```
output
bot hits with params: 3
bot hits clean: 2
budget wasted on params: 60.0 %
```

How it works:

1. Five lines mix Googlebot and bingbot hits to clean paths and several with utm and sort parameters.
2. param and clean counters start at zero.
3. For each bot hit, the path is checked for a question mark; if present param increments, otherwise clean does.
4. Both counts print, followed by the percentage of bot budget spent on parametered URLs.

Try it:

1. List the distinct parameter keys bots crawled most.
2. Include human hits and compare their parameter share to bots.
3. Treat only tracking parameters like utm as waste, not functional ones.
4. List the distinct base paths behind the parametered hits.

⚠ Common Mistakes

- Treating the fragment `#` like a query parameter is wrong; only `?` starts the query string.
- Counting human param hits dilutes the crawl-budget question, which is about bots.
- Dividing by only the param count instead of the bot total overstates the percentage.

Skills gained `query strings` `counting` `crawl budget`

→ **Where You'll Use This:** Quantifying parameter crawl makes the case for canonical tags, parameter handling and robots rules that stop bots wasting fetches on infinite filter combinations.

Mental Model

Like noticing a courier keeps delivering to the same house with slightly different apartment numbers scribbled on, wasting trips on duplicates.

Project 461 Find URL-Parameter Crawl Bloat

The Idea Behind It

When one base path explodes into many parameter variants, crawlers waste fetches on duplicates. This project strips each path at the first question mark to get its base, counts the variants per base in a Counter, then prints each base with a crawl-bloat note when it has more than one variant. Grouping by the part before the query string exposes which pages spawn too many URLs.

What You'll Build & Learn

Group crawled URLs by their base path and flag bases with many parameter variants.

- splitting a path on the question mark
- Counter of base paths
- conditional inline note

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 460: measure crawl budget spent on parameters. Project 462 changes the surface problem while preserving the underlying method.

progress check

Reading target: find url-parameter crawl bloat is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.

- Evidence: `print(f'{base}: {keys[base]} variants{note}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`note = ' <- crawl bloat' if keys[base] > 1 else ''`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{base}: {keys[base]} variants{note}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `note = ' <- crawl bloat' if keys[base] > 1 else ''`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find URL-Parameter Crawl Bloat: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, id. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: note = ' <- crawl bloat' if keys[base] > 1 else '' | Evidence: print(f'{base}: {keys[base]} variants{note}')

🛠️ Reader to Writer Bridge

1. Read: explain find url-parameter crawl bloat in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 460.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 462 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{base}: {keys[base]} variants{note}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find url-parameter crawl bloat and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Splitting each path on `?` and counting the base URL reveals which clean URLs spawned multiple parametered variants in the log.

```python
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /p?id=1 HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /p?id=2 HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /p?id=3 HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.4 - - [...] "GET /blog HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
]
keys = Counter()
for ln in lines:
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    base = path.split('?', 1)[0]
    keys[base] += 1
for base in sorted(keys):
    note = '  <- crawl bloat' if keys[base] > 1 else ''
    print(f'{base}: {keys[base]} variants{note}')
```

```
output
/blog: 1 variants
/p: 3 variants  <- crawl bloat
```

How it works:

1. Four lines crawl /p with three different id parameters plus a clean /blog.
2. An empty Counter named keys is created.
3. Each path is split on the question mark with maxsplit one, and the base before it increments the counter.
4. For each sorted base, a crawl-bloat note is appended when its variant count exceeds one before printing.

Try it:

1. Sort the output by variant count to surface the worst offender first.
2. Print the actual variant URLs under each bloated base.
3. Raise the bloat threshold so only bases with many variants are flagged.
4. Raise the bloat threshold to 3 variants.

⚠ Common Mistakes

- Splitting on `=` instead of `?` mangles paths that contain equals signs.
- Counting full parametered paths instead of bases hides that they share one page.
- Flagging bloat at one variant is too aggressive; the threshold here is more than one.

Skills gained string split | Counter | crawl bloat

→ Where You'll Use This: Finding bloated base paths reveals where faceted navigation or session parameters generate near-infinite URLs, a classic source of wasted crawl on ecommerce sites.

Mental Model

Like spotting one product that ships in fifty barely different SKUs, cluttering the catalogue with duplicates of essentially the same thing.

Project 462 Split Bandwidth Between Bots and Humans

The Idea Behind It

Bots can quietly consume a large share of bandwidth, so summing bytes separately for bot and human hits shows who is really using the pipe. This project classifies each line by its user-agent, adds its byte count to either bot_bytes or human_bytes, and prints both totals plus the bot share as a percentage. It combines the bot-detection and byte-summing patterns into a bandwidth split.

What You'll Build & Learn

Sum bytes served to bots versus humans and report the bot share of bandwidth.

- accumulating bytes by category
- bot versus human classification
- bandwidth share percentage

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 461: find url-parameter crawl bloat. Project 463 changes the surface problem while preserving the underlying method.

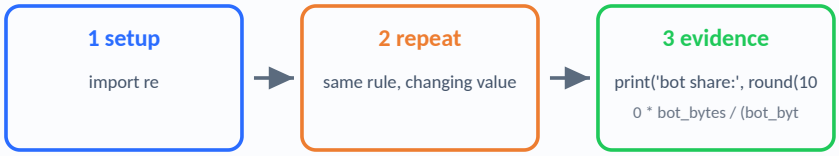
progress check

Reading target: split bandwidth between bots and humans is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('bot share:', round(100 * bot_bytes / (bot_bytes + human_bytes), 1), '%')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if any(b in ua for b in bots):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('bot share:', round(100 * bot_bytes / (bot_bytes + human_bytes), 1), '%')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if any(b in ua for b in bots):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Split Bandwidth Between Bots and Humans: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, lines, Googlebot, Firefox, Safari. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if any(b in ua for b in bots): | Evidence: print('bot share:', round(100 * bot_bytes / (bot_bytes + human_bytes),...

🛠️ Reader to Writer Bridge

1. Read: explain split bandwidth between bots and humans in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 461.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 463 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('bot share:', round(100 * bot_bytes / (bot_bytes + human_bytes), 1), '%')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of split bandwidth between bots and humans and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each hit's byte count is added to a bot or human running total based on the user-agent test, and the bot share is its fraction of the combined bytes.

```
import re
lines = [
    '1.1.1.1 - - [...] "GET /a HTTP/1.1" 200 5000 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /b HTTP/1.1" 200 2000 "-" "Firefox/120"',
```

```
    '1.1.1.3 - - [...] "GET /c HTTP/1.1" 200 3000 "-" "Googlebot/2.1"',
    '1.1.1.4 - - [...] "GET /d HTTP/1.1" 200 4000 "-" "Safari/605"',
]
bots = ('googlebot', 'bingbot')
bot_bytes = human_bytes = 0
for ln in lines:
    ua = ln.rsplit('"', 2)[1].lower()
    nbytes = int(re.search(r'" \d{3} (\d+) ', ln).group(1))
    if any(b in ua for b in bots):
        bot_bytes += nbytes
    else:
        human_bytes += nbytes
print('bot bytes:', bot_bytes)
print('human bytes:', human_bytes)
print('bot share:', round(100 * bot_bytes / (bot_bytes + human_bytes), 1), '%')
```

```
output
bot bytes: 8000
human bytes: 6000
bot share: 57.1 %
```

How it works:

1. Four lines mix Googlebot hits with Firefox and Safari hits, each with a byte size.
2. bot_bytes and human_bytes both start at zero.
3. Each pass isolates the lowercased agent and parses the bytes, then adds them to the bot or human total based on any() matching a marker.
4. Both byte totals print, followed by the bot share as a percentage.

Try it:

1. Break bot bytes down further by individual crawler name.
2. Print the totals in megabytes for readability.
3. Add the human share so both percentages display together.
4. Report the totals in KB instead of raw bytes.

⚠ Common Mistakes

- Counting hits instead of bytes answers a different question than bandwidth.
- Adding bytes to the wrong bucket on a missed bot match skews the share.
- Dividing by only the bot bytes instead of the combined total gives a nonsensical percentage.

Skills gained bandwidth | classification | percentages

→ Where You'll Use This: A bot-versus-human bandwidth split justifies rate-limiting aggressive crawlers and reveals when serving bots costs more than serving paying customers.

Mental Model

Like metering two taps off the same water main to see whether the garden hose or the kitchen is draining the supply.

Project 463 Measure a Bot's Crawl Pacing

The Idea Behind It

A polite crawler spaces its requests while a hammering one fires them seconds apart, so averaging the gaps between consecutive timestamps measures pacing. This project parses all timestamps into a list, computes the second differences between neighbours, averages them, and prints the gaps plus a polite-or-hammering verdict against a five-second cutoff. It turns a sequence of times into a rate-of-crawl signal.

What You'll Build & Learn

Measure how fast a bot crawls by averaging the seconds between its consecutive hits.

- parsing timestamps into a list
- differences between neighbours
- total_seconds for a timedelta

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 462: split bandwidth between bots and humans. Project 464 changes the surface problem while preserving the underlying method.

progress check

Reading target: measure a bot's crawl pacing is the surface task; the transferable Python move is to mark setup, rule, and output before running.

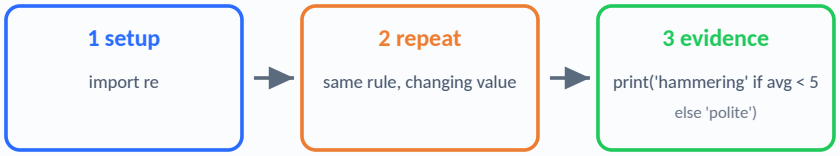
setup -> rule -> result

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('hammering' if avg < 5 else 'polite')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`print('hammering' if avg < 5 else 'polite')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('hammering' if avg < 5 else 'polite')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('hammering' if avg < 5 else 'polite')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Measure a Bot's Crawl Pacing: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, datetime, lines, Jun, Googlebot. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print('hammering' if avg < 5 else 'polite') | Evidence: print('hammering' if avg < 5 else 'polite')

🛠️ Reader to Writer Bridge

1. Read: explain measure a bot's crawl pacing in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 462.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 464 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('hammering' if avg < 5 else 'polite')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of measure a bot's crawl pacing and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Subtracting consecutive parsed timestamps yields inter-hit gaps in seconds via `total_seconds`, and the average gap classifies the crawler as polite or hammering.

```
import re
from datetime import datetime
lines = [
    '1.1.1.1 - - [22/Jun/2026:10:00:00 +0000] "GET /a HTTP/1.1" 200 1 "-"
"Googlebot/2.1"',
    '1.1.1.1 - - [22/Jun/2026:10:00:02 +0000] "GET /b HTTP/1.1" 200 1 "-"
"Googlebot/2.1"',
```

```
    '1.1.1.1 - - [22/Jun/2026:10:00:03 +0000] "GET /c HTTP/1.1" 200 1 "-"
"Googlebot/2.1"',
    '1.1.1.1 - - [22/Jun/2026:10:00:20 +0000] "GET /d HTTP/1.1" 200 1 "-"
"Googlebot/2.1"',
]
times = []
for ln in lines:
    raw = ln.split('[')[1].split()[0]
    times.append(datetime.strptime(raw, '%d/%b/%Y:%H:%M:%S'))
gaps = [(times[i] - times[i - 1]).total_seconds() for i in range(1,
len(times))]
avg = sum(gaps) / len(gaps)
print('inter-hit gaps (s):', [int(g) for g in gaps])
print('avg gap:', round(avg, 1), 's')
print('hammering' if avg < 5 else 'polite')
```

```
output
inter-hit gaps (s): [2, 1, 17]
avg gap: 6.7 s
polite
```

How it works:

1. Four Googlebot lines from one IP arrive at 10:00:00, 02, 03 and 20.
2. Each timestamp is sliced out and parsed by strptime into the times list.
3. A comprehension computes total_seconds for each adjacent pair, giving the gaps list.
4. The average gap prints with the gap list, then 'hammering' or 'polite' depending on whether the average is below five seconds.

Try it:

1. Print the smallest gap to catch the fastest burst, not just the average.
2. Group lines by IP first so each crawler's pacing is measured separately.
3. Lower the cutoff to three seconds and see how the verdict changes.
4. Print the smallest gap to find the tightest burst.

⚠ Common Mistakes

- Assuming the log is time-sorted; if not, the gaps can go negative and need sorting first.
- Using `datetime.now` is both forbidden and meaningless; the times come from the log.
- Dividing by `len(times)` instead of `len(gaps)` (one fewer) skews the average.

Skills gained datetime deltas crawl pacing

→ Where You'll Use This: Crawl pacing analysis catches bots overloading the server and informs crawl-delay settings, protecting site stability without blocking legitimate indexing.

Mental Model

Like timing the seconds between a woodpecker's taps: steady spacing is polite, a rapid drumroll means it is hammering.

Project 464 Run a Full Crawl-Waste Audit

The Idea Behind It

This capstone ties the chapter together by running every key measure in one pass over the log. It counts total hits, isolates bot traffic, tallies bot paths in a Counter, and marks each bot hit on a 404 or 3xx as wasted. It then prints total and bot hits, the wasted-crawl percentage, and the single most-crawled path, producing a compact crawl-waste audit report.

What You'll Build & Learn

Run a combined audit that reports total hits, bot hits, wasted crawl and the top path.

- combining several metrics in one loop
- wasted-crawl percentage
- most_common for the top path

Skill Roadmap

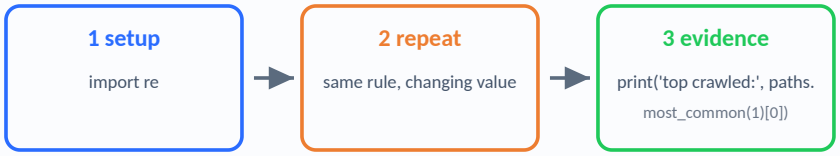
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 463: measure a bot's crawl pacing. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: run a full crawl-waste audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
- Evidence: `print('top crawled:', paths.most_common(1)[0])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
2. Apply the rule	`if any(b in ua for b in bots):`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('top crawled:', paths.most_common(1)[0])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Log-File and Server Analysis becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if any(b in ua for b in bots):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run a Full Crawl-Waste Audit: connect the surface task to the chapter skill, parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.
Names	Watch these names first: re, collections, Counter, lines, home. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if any(b in ua for b in bots): | Evidence: print('top crawled:', paths.most_common(1)[0])

🛠️ Reader to Writer Bridge

1. Read: explain run a full crawl-waste audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 463.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('top crawled:', paths.most_common(1)[0])`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run a full crawl-waste audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A single loop classifies each hit, counts bot paths, and tallies waste (bot 404s and redirects); the closing report combines coverage, waste and the top URL.

```
import re
from collections import Counter
lines = [
    '1.1.1.1 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Googlebot/2.1"',
    '1.1.1.2 - - [...] "GET /gone HTTP/1.1" 404 9 "-" "Googlebot/2.1"',
    '1.1.1.3 - - [...] "GET /old HTTP/1.1" 301 0 "-" "Googlebot/2.1"',
```

```
    '1.1.1.4 - - [...] "GET /home HTTP/1.1" 200 1 "-" "Firefox/120"',
    '1.1.1.5 - - [...] "GET /home HTTP/1.1" 200 1 "-" "bingbot/2.0"',
    '1.1.1.6 - - [...] "GET /missing HTTP/1.1" 404 9 "-" "bingbot/2.0"',
]
bots = ('googlebot', 'bingbot')
bot_hits = waste = 0
paths = Counter()
for ln in lines:
    ua = ln.rsplit('"', 2)[1].lower()
    code = int(re.search(r'" (\d{3}) ', ln).group(1))
    path = re.search(r'"\S+ (\S+) HTTP', ln).group(1)
    if any(b in ua for b in bots):
        bot_hits += 1
        paths[path] += 1
        if code == 404 or 300 <= code < 400:
            waste += 1
print('total hits:', len(lines))
print('bot hits:', bot_hits)
print('wasted crawl:', waste, '(', round(100 * waste / bot_hits, 1), '%)')
print('top crawled:', paths.most_common(1)[0])
```

```
output
total hits: 6
bot hits: 5
wasted crawl: 3 ( 60.0 %)
top crawled: ('/home', 2)
```

How it works:

1. Six lines mix Googlebot, bingbot and Firefox hits with 200, 404 and 301 codes.
2. bot_hits and waste start at zero and a Counter named paths is created.
3. For each bot hit, bot_hits and the path counter increment, and waste increments when the code is 404 or in the 3xx range.
4. It prints total hits, bot hits, the wasted percentage of bot hits, and the top path from most_common.

Try it:

1. Add total bytes served to the report for a bandwidth figure.
2. Break the wasted count into 404s versus redirects separately.
3. List the top three crawled paths instead of just the single top one.
4. Add the human-hit count to the report for completeness.

⚠ Common Mistakes

- Computing the waste percentage against total hits instead of bot hits understates it.
- Calling `most_common(1)[0]` on an empty Counter would raise; the fixture guarantees bot hits.

- Counting human 404s as crawl waste mixes user errors into the crawl-budget metric.

Skills gained audit | Counter | crawl waste

→ Where You'll Use This: A one-pass crawl-waste audit is the deliverable an SEO hands to engineering: a single report quantifying how efficiently search engines spend their budget on the site.

Mental Model

Like a single-page health summary at the end of a checkup: one sheet rolls up every test into the numbers that matter.

✅ Chapter Progress Checkpoint

What this chapter added: after 32 mini-projects, Log-File and Server Analysis should feel like a reusable Python move: parse server log lines, separate bot hits, measure crawl waste and find the most-crawled paths.

chapter checkpoint

1 Read
Open any project from Projects 433-464 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.

transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, Chapter 14 (GEO, AEO and the End-to-End Technical Audit) will reuse this chapter's reading habit with a new surface skill.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

8 questions · 4 categories · graded

2 Recall | 3 Meaning | 2 Structure | 1 Transfer

13.1.Q1 Recall MEDIUM — Turn back to Project 441 (Total the Bytes Served). Without running it, what exactly does it print?

13.1.Q2 Meaning MEDIUM — Turn back to Project 447 (Count the Unique Paths Crawled). Without running it, what exactly does it print?

13.1.Q3 Structure MEDIUM — Turn back to Project 438 (Count Requests per Path). Without running it, what exactly does it print?

13.1.Q4 Transfer MEDIUM — Turn back to Project 437 (Split Bot and Human Traffic). Without running it, what exactly does it print?

13.1.Q5 Recall EASY In a Common Log Format line, which two fields are most useful for SEO analysis?

- A IP and referer
- B Request path and status code
- C Timestamp and bytes
- D Protocol and method

13.1.Q6 Meaning EASY A line shows status 301 fetched by Googlebot. Does it count toward crawl waste?

- A No, only 404s count
- B No, 3xx is a success
- C Yes — a 3xx redirect is a fetch that returned no content, so a bot chasing it is wasted budget
- D Only if the path is in robots.txt

13.1.Q7 Structure EASY Why is splitting a log line with a plain space-based split unreliable?

- A Spaces are not allowed in logs
- B The request line and User-Agent are quoted fields that themselves contain spaces, so a naive split mis-counts the fields
- C It is too slow
- D It only works on bot lines

13.1.Q8 Meaning MEDIUM A bot made 10 fetches but only touched 4 distinct paths. What does the gap tell you, and how do you measure it in code?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 13 – Log-File and Server Analysis

13.1.Q1. `total bytes served: 5160` then `total KB: 5.04`

13.1.Q2. `unique paths: 3` then `['/blog', '/home', '/shop']`

13.1.Q3. `/blog: 2` then `/home: 3`

13.1.Q4. `bot hits: 2` then `human hits: 3`

13.1.Q5. B the path names which URL was asked for and the status says how the server answered; together they reveal crawl behaviour.

13.1.Q6. C crawl waste includes any bot fetch that returned nothing useful, and the 3xx redirect class qualifies along with 404s.

13.1.Q7. B quoted fields hold internal spaces, so positional splitting breaks; a CLF-aware regex or an rsplit on the quotes is needed.

13.1.Q8. It re-crawled pages rather than reaching new ones, so crawl breadth is low. Measure distinct paths by collecting parsed paths into a set and comparing len(set) to the total number of lines.

Chapter 14 · GEO, AEO and the End-to-End Technical Audit

Score answer-extractability and E-E-A-T signals, AI-mode readiness, and run a 32-point technical audit.

Chapter Skill Map

What this chapter adds: In GEO, AEO and the End-to-End Technical Audit, the focus is score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit. You practise it across 36 short, runnable Python projects — each one a single deliberate move from recognition to fluency, never a race through examples. *chapter arc*

What it builds on: Chapter 14 assumes the reading habits from Chapter 13: Log-File and Server Analysis. You should carry forward the habit of predicting output before trusting the page.

Where it leads: This final chapter asks you to combine earlier habits: trace, explain, modify, and test without being handed the exact next step.

1 **Projects 465-476** Recognise the pattern and read the code slowly. *name it first*	2 **Projects 477-488** Trace similar programs with less prompting and explain what changed.	3 **Projects 489-500** Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Two questions decide whether you win the search box now: **can a machine *extract* your answer, and can it *trust* you enough to cite it? ***extraction + trust = a citation***

For twenty years SEO meant *earn a blue link and hope for a click*. Answer engines changed the game. Google's AI Overviews, Bing Copilot and chat assistants no longer just list pages — they read several pages, synthesise an answer, and cite a handful of sources. GEO (generative engine optimization) and AEO (answer engine optimization) are the craft of being one of those cited sources. The blue link still exists, but the prize moved: you now compete to be *the sentence the machine lifts*, not just the tenth result.

An answer engine does two things with your page. First it extracts — it looks for a tight, self-contained chunk that resolves the query: a 30-word lead paragraph, a clean `X is a ...` definition, a numbered list, a comparison table. Second it judges trust — is there a named author with credentials, recent dates, outbound citations, schema it can parse? A page can be perfectly extractable and still be skipped because nothing signals it is worth quoting. So GEO/AEO reduces to two measurable axes: *extractability* and *trust*.

The lovely thing is that both axes are computable. You do not have to guess whether a page is answer-ready — you can measure it. The simplest GEO signal of all is the lead paragraph: answer engines reward a first paragraph that resolves the query in one tight bite, roughly 20-60 words. Too short and it is not an answer; too long and it is an essay. Here is that signal as code:

```
import re
html = ('<h1>What Is Schema Markup?</h1>'
        '<p>Schema markup is structured data added to a page that '
```

```
        'helps search engines understand its content. It uses a '
        'shared vocabulary so machines can extract facts reliably.</p>')
first = re.search(r'<p>(.*?)</p>', html).group(1)
words = len(first.split())
ok = 20 <= words <= 60
print(f'lead answer words: {words}')
print('extractable lead:', 'YES' if ok else 'NO')
```

```
output
lead answer words: 27
extractable lead: YES
```

That is the whole field in miniature. Every GEO check is a small function over the HTML that returns a score, and every audit is just a pile of such functions run together. This chapter is the capstone: it teaches the new extraction-and-trust signals, then folds them in with crawlability, indexability, status codes, on-page, canonicals, speed, structured data and hreflang from every prior chapter into one 32-point audit that grades a site end to end. *measure, score, grade, fix*

Keep one frame in mind throughout: a citation is *extraction × trust*. A page that is extractable but untrusted is quotable but unquoted; a page that is trusted but not extractable has nothing clean to lift. You need both, and the audit measures both.

GEO (GENERATIVE ENGINE OPTIMIZATION)

Optimising a page so generative AI engines (AI Overviews, Copilot, chat assistants) select, synthesise and cite its content when composing an answer — not just rank it as a link.

AEO (ANSWER ENGINE OPTIMIZATION)

The narrower discipline of structuring content so answer engines and featured snippets can extract a direct answer: question-style headings, concise lead answers, FAQ/HowTo schema.

ANSWER-EXTRACTABILITY

How easily a machine can lift a self-contained answer from a page: a concise lead paragraph, a clean definition, a list, or a table that resolves the query without further reading.

E-E-A-T

Experience, Expertise, Authoritativeness and Trust — a quality framework inferred from concrete page signals such as a named author bio, credentials, outbound citations, and an about/contact page.

CONTENT FRESHNESS

How recently a page was published or updated. Answer engines and rankers prefer recent pages for many queries; comparing an updated date to an audit date flags stale content.

TECHNICAL AUDIT

A repeatable pass that runs many independent checks (crawl, robots, sitemap, status, canonical, speed, schema, hreflang, on-page, AEO, E-E-A-T) over a site, scores them, grades the total, and lists what to fix.

Extraction: the four shapes a machine can lift

Answer engines do not paraphrase a whole article — they look for a chunk in a *shape* they can quote. There are four shapes worth engineering for: a concise lead paragraph (the direct answer up top), a definition sentence (X `is a ...`), a list (steps or bullets), and a table (a comparison). Classifying each block by its shape predicts how it can be reused. If a block is none of these — just prose — it is far less likely to be extracted.

```
import re
blocks = {
    'definition': '<p>Crawl budget is the number of pages Googlebot fetches.</p>',
    'steps':      '<ol><li>Open Search Console</li><li>Submit sitemap</li></ol>',
    'comparison': '<table><tr><th>301</th><th>302</th></tr></table>',
    'plain':      '<p>We think SEO is great and you should try it sometime.</p>',
}
for name, block in blocks.items():
    if re.search(r'<ol>|<ul>', block):
        fmt = 'list/steps'
    elif '<table' in block:
        fmt = 'table/comparison'
    elif re.match(r'<p>\w+( \w+)? (is|are) (a |an |the )', block):
        fmt = 'definition'
    else:
        fmt = 'prose'
    print(f'{name}: {fmt}')
```

```
output
definition: definition
steps: list/steps
comparison: table/comparison
plain: prose
```

Pair an extractable shape with a *question-style heading* and you have the recipe for a featured snippet or an answer box. An interrogative H1 like "How Long Should A Title Tag Be?" tells the engine exactly which query the page

resolves; a substantial first paragraph then supplies the answer. That heading-plus-answer pattern is the single highest-leverage AEO move.

Passage ranking: the engine quotes one sentence, not the page

When an engine answers a query it often pulls a *single passage* — the one sentence on the page that best resolves the question. A crude but faithful model of how it chooses is `term overlap`: tokenise the question, tokenise each candidate passage, and pick the passage that shares the most terms. Write content as self-contained sentences and you give the engine clean passages to choose from; bury the answer mid-paragraph and it has nothing to lift.

```
import re
question = 'what is canonical url'
passages = [
    'A canonical URL is the single preferred address you tell '
    'search engines to index for duplicate pages.',
    'Sitemaps list the URLs you want crawled.',
    'Redirects send users from an old URL to a new one.',
]
qterms = set(re.findall(r'[a-z]+', question.lower()))
best, score = None, -1
for p in passages:
    overlap = len(qterms & set(re.findall(r'[a-z]+', p.lower())))
    if overlap > score:
        best, score = p, overlap
print('best passage:', best)
print('term overlap:', score)
```

```
output
best passage: A canonical URL is the single preferred address you tell search engines to
index for duplicate pages.
term overlap: 3
```

The same overlap idea, run the other way, audits *coverage*: map every question your audience asks to the heading that best covers it, and any question with no matching heading is a `content gap` — a query you cannot win because you never wrote the answer.

Trust: turning E-E-A-T from a vibe into a number

Experience, Expertise, Authoritativeness, Trust sounds unmeasurable, but engines infer it from concrete page signals, so you can too. Sum a checklist: is there an author bio? author credentials? at least a couple of outbound citations? published and updated dates? an about page and a contact page? Each present signal is a point. A fuzzy quality concept becomes a score out of seven.

```
signals = {
    'author_bio': True, 'author_credentials': True,
    'outbound_citations': 4, 'published_date': True,
    'updated_date': False, 'about_page': True, 'contact_page': False,
```

```
}
score = 0
score += 1 if signals['author_bio'] else 0
score += 1 if signals['author_credentials'] else 0
score += 1 if signals['outbound_citations'] >= 2 else 0
score += 1 if signals['published_date'] else 0
score += 1 if signals['updated_date'] else 0
score += 1 if signals['about_page'] else 0
score += 1 if signals['contact_page'] else 0
print(f'E-E-A-T score: {score}/7')
print('grade:', 'strong' if score >= 6 else 'moderate' if score >= 4 else 'weak')
```

```
output
E-E-A-T score: 5/7
grade: moderate
```

Two of those signals deserve their own checks. Freshness: compare each page's updated date to a fixed audit date and flag anything older than a year — answer engines distrust stale pages. Citations: count the outbound links to authoritative `https://` domains, because linking out to real sources is itself a trust cue. Here is the freshness flag:

```
from datetime import datetime
AS_OF = '2026-06-22'
pages = {'/guide-a': '2026-05-01', '/guide-b': '2024-11-15', '/guide-c': '2026-06-10'}
asof = datetime.strptime(AS_OF, '%Y-%m-%d')
for path in sorted(pages):
    updated = datetime.strptime(pages[path], '%Y-%m-%d')
    age_days = (asof - updated).days
    state = 'STALE' if age_days > 365 else 'fresh'
    print(f'{path}: {age_days}d old -> {state}')
```

```
output
/guide-a: 52d old -> fresh
/guide-b: 584d old -> STALE
/guide-c: 12d old -> fresh
```

AEO schema: handing the engine ready-made Q&A

The cleanest extractable shape of all is structured data that *is* a question and answer. `FAQPage`, `HowTo` and `QAPage` JSON-LD give an answer engine machine-readable question/answer pairs — no extraction guesswork required. Scanning a page's `@type` values for those tells you whether it is AEO-equipped. (Validity matters too — the schema must parse and declare a `@type`, which is the structured-data check we reuse in the audit.)

```
import re
html = ('<script type="application/ld+json">{"@type":"FAQPage"}</script>'
```

```
        '<script type="application/ld+json">{"@type":"Article"}</script>'
        '<script type="application/ld+json">{"@type":"HowTo"}</script>')
types = re.findall(r'"@type"\s*:\s*"(\w+)"', html)
aeo_types = {'FAQPage', 'HowTo', 'QAPage'}
present = sorted(set(types) & aeo_types)
print('schema types:', sorted(set(types)))
print('AEO-relevant:', present)
print('AEO-ready:', 'YES' if present else 'NO')
```

```
output
schema types: ['Article', 'FAQPage', 'HowTo']
AEO-relevant: ['FAQPage', 'HowTo']
AEO-ready: YES
```

The end-to-end audit: every chapter, one score

Now the capstone move. Every prior chapter contributed an independent check — robots reachable, sitemap valid, canonical self-referencing, key pages return 200, Core Web Vitals in the green, hreflang reciprocal with an x-default, schema parses, title and meta lengths sane. Add this chapter's GEO/AEO signals — concise lead answer, AEO schema, author bio, freshness — and you have `32 checks in four groups of eight`: crawlability, indexability+status, performance, and content+AEO+E-E-A-T. Run them all over one site, count the passes, and map the percentage to a letter grade.

```
s = {
    # crawlability (8)
    'robots_ok': True, 'no_disallow_all': True, 'crawl_delay_ok': True,
    'sitemap_present': True, 'sitemap_size_ok': True, 'sitemap_valid': True,
    'no_orphans': False, 'internal_links_ok': True,
    # indexability + status (8)
    'canonical_self': True, 'no_noindex_key': True, 'https': True, 'all_200': True,
    'redirects_short': True, 'hreflang_reciprocal': True, 'x_default': False,
'no_broken': True,
    # performance (8)
    'lcp_ok': True, 'inp_ok': True, 'cls_ok': False, 'mobile_ok': True,
    'compressed': True, 'render_block_ok': True, 'weight_ok': True, 'ttfb_ok':
True,
    # content + AEO + E-E-A-T (8)
    'title_ok': True, 'meta_ok': True, 'h1_ok': True, 'schema_valid': True,
    'lead_answer_ok': True, 'aeo_schema': True, 'author_bio': True, 'fresh': False,
}
failed = [c for c in s if not s[c]]
score = len(s) - len(failed)
pct = round(100 * score / 32)
grade = 'A' if pct >= 90 else 'B' if pct >= 80 else 'C' if pct >= 70 else 'D' if
pct >= 60 else 'F'
print('=== 32-POINT TECHNICAL SEO AUDIT ===')
```

```
print(f'score: {score}/32 ({pct}%)  grade: {grade}')
print('failures:', failed)
```

```
output
=== 32-POINT TECHNICAL SEO AUDIT ===
score: 28/32 (88%)  grade: B
failures: ['no_orphans', 'x_default', 'cls_ok', 'fresh']
```

A score and a list of failing names is not yet useful — a stakeholder cannot act on `cls_ok: False`. The last step maps each failure to a fix, turning the audit into a remediation plan; and run over time, diffing today's audit against a baseline surfaces regressions (checks that broke) and wins (checks you fixed). That loop — measure, grade, fix, re-measure — is what makes an audit a tool rather than a one-off report.

an audit you cannot act on and re-run is just a screenshot

Example 1

A page has a question H1 "What Is A 301 Redirect?", a 24-word lead paragraph answering it, a bulleted list of redirect types, supporting detail below, but no QAPage schema. Score its answer-box readiness on the 5-point checklist, and say whether it can win a featured snippet.

SOLUTION

1. Walk the five checks: question-style heading yes; concise answer up top yes (24 words, under 60); answer under 60 words yes; supporting detail below yes; QAPage schema no.
2. Four of five pass, so readiness is 4/5 — strong but not perfect.
3. Featured snippets are drawn from a 40-60 word paragraph, a list, or a table. The page has a list, so it is snippet-eligible even without the schema; the missing QAPage only costs the cleaner answer-box treatment.

Answer-box readiness 4/5; yes, it can win a snippet because the bulleted list is an extractable structure.

a list or table alone is enough to be snippet-eligible — schema is the bonus, not the gate

Example 2

Run a weighted AI-mode readiness score for a page with clear_headings=yes, schema=yes, concise_answers=yes, fresh=no, cited_sources=yes, using weights headings 2, schema 2, concise 3, fresh 1, citations 2. Is it ready, and what is the top gap?

SOLUTION

1. Sum the weights of the *passing* signals: headings 2 + schema 2 + concise 3 + citations 2 = `9`. The only failing signal is freshness (weight 1).
2. Total possible weight is 2+2+3+1+2 = 10, so the score is 9/10 = `90%`.
3. A common readiness bar is 70%; 90% clears it comfortably, so the verdict is *ready*. The single missing signal — freshness — is the top gap to fix.

AI-mode readiness 9/10 (90%), verdict ready; the top gap is freshness.

weighting matters: a concise answer is worth 3, freshness only 1, so fix the heavy signals first

Example 3

A site scores 28/32 on the technical audit with failures no_orphans, x_default, cls_ok and fresh, plus a GEO/AEO sub-score of 4/5 (missing: fresh). Combine them into one scorecard and give the letter grade and the highest-priority fix.

SOLUTION

1. Combine the two scores: technical 28 + GEO/AEO 4 = `32` out of (32 + 5) = 37 total.
2. Percentage is 32/37 = 86%, which lands in the 80-89 band → grade `B`.
3. Freshness fails in *both* halves (it is one of the four technical failures and the one GEO gap), so it is the single fix that lifts two scores at once — the highest-priority remediation.

Combined 32/37 (86%), grade B; refresh the stale content and bump its lastmod date first, because freshness is the failure that appears on both scorecards.

fix the failure that shows up in more than one group first — it has the best score-per-effort

⚡ Mental-Math Shortcut

A citation is extraction times trust — and an audit is just a pile of small checks. To be cited, a page must offer a *clean shape to lift* (a concise lead, a definition, a list, a table) and *reasons to be trusted* (named author, recent date, real sources, parseable schema); missing either factor zeroes the product. And whenever a quality idea feels unmeasurable — E-E-A-T, "answer-readiness", site health — turn it into a checklist, sum the passes, grade the percentage, and map each failure to a fix. Measure, score, grade, remediate, repeat.

⚠ Common Mistakes

- Treating GEO/AEO as new keyword stuffing — packing question phrases everywhere instead of writing one clean, self-contained answer in an extractable shape near the top.
- Burying the answer. A correct answer in paragraph six loses to a worse answer in the first sentence, because the engine extracts the passage it can lift, not the best one it could find with effort.
- Adding FAQPage/HowTo schema but never validating it — invalid JSON-LD or a missing `@type` means the engine ignores the markup entirely, so the AEO check should require the schema to parse.
- Auditing once and filing the report. The value is in re-running it: diffing against a baseline catches regressions, and mapping failures to fixes is what makes the score actionable rather than decorative.
- Optimising extraction while ignoring trust (or vice versa). A perfectly structured page with no author, no dates and no citations is quotable but rarely quoted; a trusted page with no clean passage has nothing to lift.

This chapter contains 36 hands-on mini-projects (Projects 465–500). Type each one out, run it, and experiment.

Project 465 Score a Page's Answer-Extractability Lead

The Idea Behind It

Generative engines reward a page whose opening paragraph resolves the query in one tight pass. This project pulls the first paragraph out of raw HTML with a regular expression, counts its words after splitting on whitespace, and treats a lead between twenty and sixty words as the sweet spot for extraction. The single boolean comparison turns a vague editorial instinct about a good intro into a hard, reproducible answer-extractability signal you can run over any page.

What You'll Build & Learn

Measure whether a page's first paragraph is a concise, on-topic answer that an AI engine can lift verbatim.

- Extract the lead paragraph with a regex
- Word-count a string after splitting
- Range checks express editorial rules

Skill Roadmap

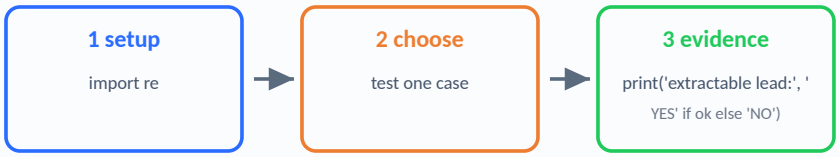
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 466 changes the surface problem while preserving the underlying method. *progress check*

Reading target: score a page's answer-extractability lead is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('extractable lead:', 'YES' if ok else 'NO')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print('extractable lead:', 'YES' if ok else 'NO')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('extractable lead:', 'YES' if ok else 'NO')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('extractable lead:', 'YES' if ok else 'NO')`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Score a Page's Answer-Extractability Lead: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, h1, What, Is. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: print('extractable lead:', 'YES' if ok else 'NO') | Evidence: print('extractable lead:', 'YES' if ok else 'NO')

🛠️ Reader to Writer Bridge

1. Read: explain score a page's answer-extractability lead in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then start by making this chapter's core idea concrete. *read with a pencil*
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 466 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('extractable lead:', 'YES' if ok else 'NO')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of score a page's answer-extractability lead and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Pulling the first `<p>` and checking its word count against a 20-60 band approximates whether the lead is a snippet-sized answer rather than a vague intro.

```
import re
html = '<h1>What Is Schema Markup?</h1><p>Schema markup is structured data
added to a page that helps search engines understand its content. It uses a
shared vocabulary so machines can extract facts reliably.</p><p>More details
follow below.</p>'
first = re.search(r'<p>(.*?)</p>', html).group(1)
words = len(first.split())
ok = 20 <= words <= 60
print(f'lead answer words: {words}')
print('extractable lead:', 'YES' if ok else 'NO')
```

```
output
lead answer words: 27
extractable lead: YES
```

How it works:

1. An HTML string holds a heading plus two paragraphs of definition-style copy about schema markup.
2. re.search with a non-greedy pattern captures the text inside the first opening and closing paragraph tag.
3. Splitting that captured text on whitespace and taking its length yields the lead word count.
4. The boolean ok holds True only when the count sits between twenty and sixty inclusive.
5. Two print lines report the raw word count and a plain YES or NO verdict on extractability.

Try it:

1. Loop over every paragraph and report the position of the first one that lands in the twenty-to-sixty window.
2. Strip nested inline tags like anchors before counting so markup does not inflate the word total.
3. Make the lower and upper bounds parameters so different engines can use their own thresholds.
4. Lower the upper bound to 40 words and see this lead still pass.

⚠ Common Mistakes

- A greedy `<p>(.*)</p>` would span multiple paragraphs; the lazy `.*?` stops at the first close tag.
- Counting characters instead of words makes the 20-60 band meaningless for answer length.

- If the page opens with a hero image or nav `<p>`, the first paragraph may not be the real lead.

Skills gained regex extraction | word counting | range validation | GEO signals

→ Where You'll Use This: Auditing whether blog posts and FAQ answers open with a quotable lead is now table stakes for AI Overviews and featured-snippet work, and this check scales across a whole content library.

Mental Model

Like a newspaper lede: the first sentence has to deliver the story before the reader, or the engine, decides whether to keep reading.

Project 466 Detect a Direct Answer Near the Top

The Idea Behind It

AI Overviews and featured snippets favour pages that pair an interrogative heading with an immediate, substantial answer. The code reads an H1 and the body paragraphs, ignoring a sponsored ad block, and checks two things independently: whether the heading ends with a question mark, and whether the first real paragraph carries at least eight words. Separating the heading test from the answer test lets you see exactly which half of the question-and-answer contract a page is honouring.

What You'll Build & Learn

Confirm a page poses a question in its heading and resolves it with a meaningful paragraph right below.

- Detect question headings via a suffix test
- Find the first substantial body paragraph
- Combine two independent boolean signals

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 465: score a page's answer-extractability lead. Project 467 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect a direct answer near the top is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

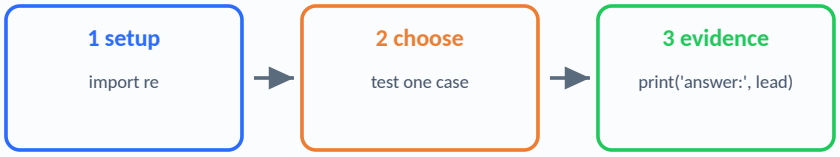
Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as `data -> transformation -> evidence`, not as isolated

library calls.

- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('answer:', lead)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`lead = paras[0] if paras else ''`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print('answer:', lead)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `lead = paras[0] if paras else ''`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Direct Answer Near the Top: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, h1, How, Long. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; classes/records group responsibilities; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: lead = paras[0] if paras else '' | Evidence: print('answer:', lead)

🛠️ Reader to Writer Bridge

1. Read: explain detect a direct answer near the top in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 465.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 467 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('answer:', lead)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a direct answer near the top and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A heading ending in `?` plus a non-trivial first paragraph (>= 8 words) is a strong cue that the page leads with a direct answer.

```
import re
html = '<h1>How Long Should A Title Tag Be?</h1><div class="ad">Sponsored</div>
<p>A title tag should be about 50 to 60 characters so it does not get truncated
in search results.</p>'
# strip tags, find first substantial paragraph text near the top
paras = re.findall(r'<p>(.*?)</p>', html)
lead = paras[0] if paras else ''
heading = re.search(r'<h1>(.*?)</h1>', html).group(1)
is_q = heading.strip().endswith('?')
direct = bool(lead) and len(lead.split()) >= 8
print('heading is a question:', is_q)
print('direct answer present:', direct)
print('answer:', lead)
```

```
output
heading is a question: True
direct answer present: True
answer: A title tag should be about 50 to 60 characters so it does not get truncated
in search results.
```

How it works:

1. re.findall gathers every paragraph, and the first one becomes the candidate lead while an ad div is skipped.
2. A separate re.search captures the H1 text so the heading can be examined on its own.
3. endswith on the stripped heading sets is_q True when the title is phrased as a question.
4. The direct flag requires a non-empty lead whose split length is at least eight words.
5. Three prints expose the question verdict, the direct-answer verdict, and the actual answer text.

Try it:

1. Accept headings ending in other interrogatives by also matching leading words like how, what, or why.
2. Raise the answer threshold and require the lead to repeat a keyword from the heading.
3. Walk all headings and pair each with the nearest following paragraph instead of only the H1.
4. Strip the trailing `?` from the heading and watch `is a question` go False.

⚠ Common Mistakes

- An ad or banner `<div>` between the heading and the answer is skipped because only `<p>` text is collected.
- Requiring exactly one `<p>` would miss pages with a short lead followed by detail.

- A heading phrased as a statement, not a question, fails `endswith('?')` even if the page answers well.

Skills gained heading detection | paragraph parsing | boolean logic | AEO

→ Where You'll Use This: When optimising help-centre and how-to pages for answer boxes, this verifies the basic structural pattern engines look for before deciding a page deserves a direct-answer treatment.

Mental Model

Think of a quiz card: the question on the front is useless unless the answer on the back is right there and complete.

Project 467 Detect FAQ and HowTo Schema for AEO

The Idea Behind It

Answer engines love structured data that hands them ready-made question-and-answer pairs, so FAQPage, HowTo and QAPage markup is worth detecting directly. This snippet scans concatenated JSON-LD scripts with a regex that captures every @type value, then intersects the discovered set with a known set of AEO-relevant types. Set intersection is the elegant trick here: it cleanly keeps only the markup that matters for answer optimisation and ignores ordinary Article or other types.

What You'll Build & Learn

Scan a page's JSON-LD for the schema types that feed answer engines and flag whether any are present.

- Capture schema @type values with regex
- Use set intersection to filter
- Distinguish AEO-relevant markup from generic

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 466: detect a direct answer near the top. Project 468 changes the surface problem while preserving the underlying method.

progress check

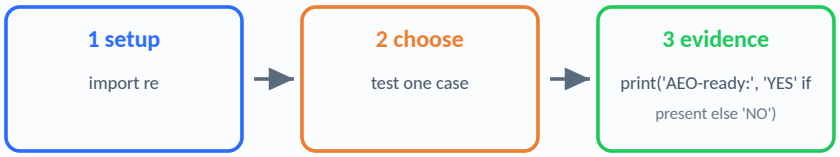
Reading target: detect faq and howto schema for aeo is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.

- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('AEO-ready:', 'YES' if present else 'NO')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print('AEO-ready:', 'YES' if present else 'NO')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('AEO-ready:', 'YES' if present else 'NO')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('AEO-ready:', 'YES' if present else 'NO')`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Detect FAQ and HowTo Schema for AEO: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, script, type, application. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print('AEO-ready:', 'YES' if present else 'NO') | Evidence: print('AEO-ready:', 'YES' if present else 'NO')

🛠️ Reader to Writer Bridge

1. Read: explain detect faq and howto schema for aeo in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 466.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 468 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('AEO-ready:', 'YES' if present else 'NO')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect faq and howto schema for aeo and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Extracting every `@type` with a regex and intersecting with the AEO-relevant set tells you whether the page exposes answer-friendly schema.

```
import re
html = '''<script type="application/ld+json">{"@type":"FAQPage"}</script>
<script type="application/ld+json">{"@type":"Article"}</script>
<script type="application/ld+json">{"@type":"HowTo"}</script>'''
types = re.findall(r'"@type"\s*:\s*"(\w+)"', html)
aeo_types = {'FAQPage', 'HowTo', 'QAPage'}
present = sorted(set(types) & aeo_types)
print('schema types:', sorted(set(types)))
print('AEO-relevant:', present)
print('AEO-ready:', 'YES' if present else 'NO')
```

```
output
schema types: ['Article', 'FAQPage', 'HowTo']
AEO-relevant: ['FAQPage', 'HowTo']
AEO-ready: YES
```

How it works:

1. A triple-quoted string holds three JSON-LD script blocks declaring FAQPage, Article and HowTo types.
2. re.findall pulls every quoted value that follows an @type key into a list of type names.
3. The found types are deduplicated into a set and intersected with the aeo_types reference set.
4. Sorting that intersection produces a stable, alphabetised list of the AEO-relevant types present.
5. A final print reports YES when any AEO type was matched and NO otherwise.

Try it:

1. Parse each script block with json.loads instead of regex so nested and array-valued @type entries are handled.
2. Report a count of how many distinct AEO types appear rather than a single yes-or-no.
3. Add Recipe and QAPage variants and warn when an Article page is missing any answer schema entirely.
4. Add a `"Recipe"` type and confirm it is not counted as AEO-relevant.

⚠ Common Mistakes

- A regex over raw text ignores JSON nesting; a `@type` inside an unrelated object still matches.
- Sorting is required because a set's print order is otherwise nondeterministic.
- Matching `"FAQ"` as a substring would wrongly catch `FAQPage` and vice versa; match the whole token.

Skills gained JSON-LD parsing | set operations | schema detection | AEO

→ Where You'll Use This: Verifying that priority pages actually emit FAQ or HowTo markup is a recurring audit task, since this schema is what wins expandable answer results and AI citations.

⚡ Mental Model

Schema is a label printed in a language only machines read; this check makes sure the most useful labels are actually on the box.

Project 468 Find the Passage That Answers a Question

💡 The Idea Behind It

Passage ranking lets an engine surface the single best sentence on a page rather than the whole document, and this project models that selection with a term-overlap score. It tokenises the question into a set of lowercase words, then for each candidate passage counts how many question terms reappear, keeping a running best. The loop is a tiny argmax: it never sorts, it simply remembers the highest overlap seen so far, which mirrors how a retriever picks one answer span from many.

What You'll Build & Learn

Pick the passage on a page that best answers a question by scoring shared terms against each candidate.

- Tokenise text into word sets
- Score relevance by term overlap
- Track a running maximum in a loop

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 467: detect faq and howto schema for aeo. Project 469 changes the surface problem while preserving the underlying method.

progress check

Reading target: find the passage that answers a question is the surface task; the transferable Python move is to mark setup, rule, and output before running.

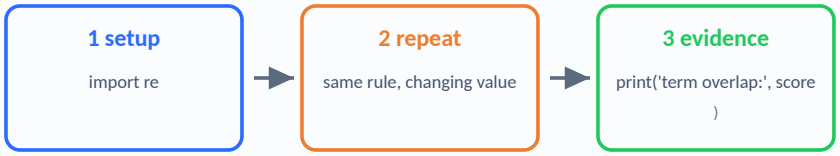
setup -> rule -> result

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('term overlap:', score)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if overlap > score:`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('term overlap:', score)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if overlap > score:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Find the Passage That Answers a Question: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, question, what, is, canonical. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if overlap > score: | Evidence: print('term overlap:', score)

🛠️ Reader to Writer Bridge

1. Read: explain find the passage that answers a question in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 467.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 469 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('term overlap:', score)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of find the passage that answers a question and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Tokenising both the question and each passage and counting shared words gives a crude relevance score; the highest-overlap passage is the likely answer.

```
import re
question = 'what is canonical url'
passages = [
    'A canonical URL is the single preferred address you tell search engines to
index for duplicate pages.',
    'Sitemaps list the URLs you want crawled.',
    'Redirects send users from an old URL to a new one.',
```

```
]
qterms = set(re.findall(r'[a-z]+', question.lower()))
best, score = None, -1
for p in passages:
    pterms = set(re.findall(r'[a-z]+', p.lower()))
    overlap = len(qterms & pterms)
    if overlap > score:
        best, score = p, overlap
print('best passage:', best)
print('term overlap:', score)
```

```
output
best passage: A canonical URL is the single preferred address you tell search
engines to index for duplicate pages.
term overlap: 3
```

How it works:

1. A question string and a list of three candidate passages about SEO concepts are defined upfront.
2. re.findall extracts lowercase alphabetic tokens from the question into a set called qterms.
3. Each passage is tokenised the same way, and the intersection size becomes its overlap score.
4. When a passage beats the current best score, best and score are updated to that passage.
5. The code prints the winning passage text and its term-overlap count.

Try it:

1. Weight rarer question terms more heavily so common words do not dominate the overlap score.
2. Break ties by preferring the shorter passage, which is usually the more quotable one.
3. Return the top three passages with their scores instead of only the single winner.
4. Add a stop-word filter so only content words contribute to overlap.

⚠ Common Mistakes

- Without lowercasing, `URL` and `url` count as different terms and miss the match.
- Stop words like `is` inflate overlap; a real ranker would remove them first.
- Ties are broken by whichever passage is seen first, which depends on list order.

Skills gained tokenisation | overlap scoring | argmax loop | passage ranking

→ Where You'll Use This: When diagnosing why a page is not cited for a query, scoring its passages against the target question shows whether any single sentence is a strong enough answer to be lifted.

Mental Model

It is a spotlight operator scanning a stage: whichever performer shares the most cues with the script gets the light.

Project 469 Match Questions to the Best Heading

The Idea Behind It

Mapping the questions people ask to the headings a page actually offers reveals how well content is organised around real intent. The code defines a keyword helper that tokenises a string and drops common stop words, then for each question uses max with a key function to pick the heading sharing the most meaningful words. Filtering stop words first is what makes the match honest, because it stops glue words like the or your from creating false overlaps between unrelated phrases.

What You'll Build & Learn

Match each user question to the page heading that best covers it and flag questions with no good match.

- Strip stop words before matching
- Use max with a key function
- Surface content gaps from zero matches

Skill Roadmap

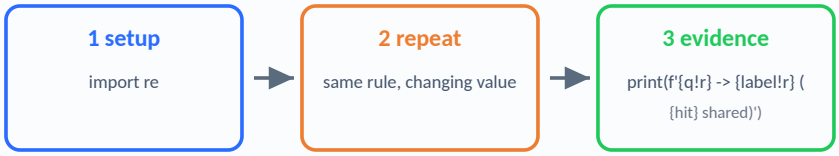
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 468: find the passage that answers a question. Project 470 changes the surface problem while preserving the underlying method. *progress check*

Reading target: match questions to the best heading is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'{q!r} -> {label!r} ({hit} shared)')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`return {w for w in re.findall(r'[a-z]+', s.lower()) if w not in stop}`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{q!r} -> {label!r} ({hit} shared)')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return {w for w in re.findall(r'[a-z]+', s.lower()) if w not in stop}`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Match Questions to the Best Heading: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, questions, How, submit, sitemap. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: return {w for w in re.findall(r'[a-z]+', s.lower()) if w not in stop} | Evidence: print(f'{q!r} -> {label!r} ({hit} shared)')

🛠️ Reader to Writer Bridge

1. Read: explain match questions to the best heading in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 468.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 470 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{q!r} -> {label!r} ({hit} shared)')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of match questions to the best heading and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

After stripping question words, the heading with the most shared content terms is the best topical match; zero shared terms means no heading covers that question.

```
import re
questions = ['How do I submit a sitemap?', 'What is crawl budget?', 'When does
Google recrawl?']
headings = ['Submitting Your Sitemap to Search Console', 'Understanding Crawl
```

```
Budget', 'How Often Google Recrawls Pages']
def keywords(s):
    stop =
{'how','do','i','a','what','is','when','does','the','to','your','and','often'}
    return {w for w in re.findall(r'[a-z]+', s.lower()) if w not in stop}
for q in questions:
    qk = keywords(q)
    best = max(headings, key=lambda h: len(qk & keywords(h)))
    hit = len(qk & keywords(best))
    label = best if hit else 'NO MATCH'
    print(f'{q!r} -> {label!r} ({hit} shared)')
```

```
output
'How do I submit a sitemap?' -> 'Submitting Your Sitemap to Search Console' (1
shared)
'What is crawl budget?' -> 'Understanding Crawl Budget' (2 shared)
'When does Google recrawl?' -> 'How Often Google Recrawls Pages' (1 shared)
```

How it works:

1. Parallel lists hold three user questions and three section headings drawn from a hypothetical guide.
2. The keywords helper lowercases, extracts alphabetic tokens, and removes a small stop-word set.
3. For each question, max scans the headings and returns the one with the largest keyword intersection.
4. If the best heading shares zero keywords, the label becomes the literal string NO MATCH.
5. Each line prints the question, its chosen heading or the gap marker, and the shared-word count.

Try it:

1. Expand the stop-word list or load it from a file so the matcher generalises across topics.
2. Add a minimum-overlap threshold so a weak single-word match is also reported as a gap.
3. Stem words like recrawl and recrawls to the same root so morphological variants still match.
4. Add a question with no matching heading and watch it report NO MATCH.

⚠ Common Mistakes

- A weak stop-word list leaves words like `often` to skew the overlap counts.
- `recrawl` and `recrawls` differ as tokens, so the match here rides on the shared word `google`.
- `max` over headings returns the first on a tie, so heading order can decide close calls.

Skills gained stop-word filtering | keyword matching | gap analysis | content mapping

→ Where You'll Use This: Content teams use this kind of question-to-heading mapping to audit whether a page structures its sections around the actual queries it is meant to rank for, exposing missing sections.

Mental Model

Like checking a book's chapter titles against a reader's questions: a question with no matching chapter is a hole in the table of contents.

Project 470 Test Featured-Snippet Eligibility

The Idea Behind It

Featured snippets are extracted from one of three shapes, a forty-to-sixty word paragraph, a list, or a table, so eligibility is really a question of format presence. The code measures the first paragraph's word count for the paragraph rule, then checks the markup for list and table tags, and declares the page eligible if any one of those conditions holds. The logical OR is the key idea: a page does not need every format, just one extractable structure, to qualify for a snippet.

What You'll Build & Learn

Estimate whether a page can win a featured snippet by checking for a tight paragraph, a list, or a table.

- Test multiple eligibility paths with OR
- Detect list and table markup
- Apply a paragraph word-count band

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 469: match questions to the best heading. Project 471 changes the surface problem while preserving the underlying method. *progress check*

Reading target: test featured-snippet eligibility is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as `data -> transformation -> evidence`, not as isolated library calls.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.

- Evidence: `print('snippet-eligible:', 'YES' if eligible else 'NO')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print('snippet-eligible:', 'YES' if eligible else 'NO')`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print('snippet-eligible:', 'YES' if eligible else 'NO')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('snippet-eligible:', 'YES' if eligible else 'NO')`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Test Featured-Snippet Eligibility: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, redirect, is, permanent. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: print('snippet-eligible:', 'YES' if eligible else 'NO') | Evidence: print('snippet-eligible:', 'YES' if eligible else 'NO')

🛠️ Reader to Writer Bridge

1. Read: explain test featured-snippet eligibility in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 469.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 471 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('snippet-eligible:', 'YES' if eligible else 'NO')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of test featured-snippet eligibility and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A page qualifies if it has a paragraph in the snippet word band OR a list OR a table; any one extractable format is enough.

```
import re
html = '<p>A 301 redirect is a permanent server instruction that points one URL
to another and passes ranking signals to the new address for good.</p><ul>
<li>301 permanent</li><li>302 temporary</li></ul>'
para = re.search(r'<p>(.*?)</p>', html).group(1)
wc = len(para.split())
has_list = '<ul>' in html or '<ol>' in html
has_table = '<table' in html
para_ok = 40 <= wc <= 60
eligible = para_ok or has_list or has_table
print(f'paragraph words: {wc} (40–60 ok: {para_ok})')
print(f'has list: {has_list} | has table: {has_table}')
print('snippet-eligible:', 'YES' if eligible else 'NO')
```

```
output
paragraph words: 24 (40–60 ok: False)
has list: True | has table: False
snippet-eligible: YES
```

How it works:

1. An HTML string pairs a redirect definition paragraph with a short two-item unordered list.
2. re.search captures the first paragraph and its split length gives the word count wc.
3. para_ok is True only inside the forty-to-sixty band, while has_list and has_table test for tags.
4. eligible becomes True if the paragraph rule passes or either structural format is present.
5. Prints report the paragraph stats, the list and table flags, and the final eligibility verdict.

Try it:

1. Distinguish ordered from unordered lists and report which snippet type each format would feed.
2. Count how many qualifying structures exist so a richer page scores higher than a bare one.
3. Add a heading-plus-paragraph rule so definition snippets near an H2 also count as eligible.
4. Lengthen the paragraph into the 40-60 band and watch both routes qualify.

⚠ Common Mistakes

- The paragraph here is too short, so eligibility rides entirely on the list being present.
- Checking only `<ol>` misses unordered `<ul>` lists that snippets also use.
- An inline `<table>` used purely for layout would be a false positive for a data table.

Skills gained format detection | boolean OR | snippet eligibility | HTML parsing

→ Where You'll Use This: Before chasing a featured snippet for a target query, this quickly confirms the page even offers an extractable format, saving effort on pages that present only loose prose.

Mental Model

Snippets are like vending-machine slots shaped for paragraphs, lists, or tables; your content only needs to fit one slot to drop into the tray.

Project 471 Score E-E-A-T Signals From a Dict

The Idea Behind It

Experience, expertise, authoritativeness and trust feel abstract until you reduce them to concrete page signals you can count. This project stores seven such signals in a dictionary, then adds one point for each that passes its own rule, including a threshold check that outbound citations must reach at least two. Turning a fuzzy quality concept into a seven-point integer, then bucketing that integer into strong, moderate or weak, is exactly how an auditor makes E-E-A-T discussable with stakeholders.

What You'll Build & Learn

Convert a checklist of concrete trust signals into a single E-E-A-T score and a plain quality grade.

- Model qualitative signals as a dict
- Score with conditional point additions
- Bucket a score into named grades

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 470: test featured-snippet eligibility. Project 472 changes the surface problem while preserving the underlying method.

progress check

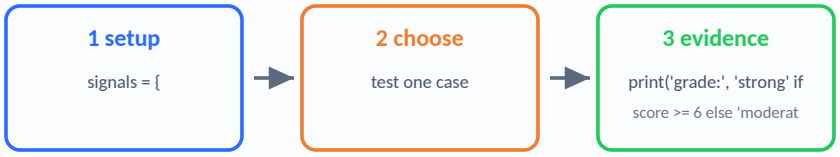
Reading target: score e-e-a-t signals from a dict is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `signals = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('grade:', 'strong' if score >= 6 else 'moderate' if score >= 4 else 'weak')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`signals = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`score += 1 if signals['author_bio'] else 0`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. slow line
3. Check the evidence	`print('grade:', 'strong' if score >= 6 else 'moderate' if score >= 4 else 'weak')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `score += 1 if signals['author_bio'] else 0`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Score E-E-A-T Signals From a Dict: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: signals, author_bio, author_credentials, outbound_citations, published_date. Names are the reader's map of roles and responsibilities.
Flow	conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: signals = { | Core: score += 1 if signals['author_bio'] else 0 | Evidence: print('grade:', 'strong' if score >= 6 else 'moderate' if score >= 4 el...

🛠️ Reader to Writer Bridge

1. Read: explain score e-e-a-t signals from a dict in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 470.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 472 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('grade:', 'strong' if score >= 6 else 'moderate' if score >= 4 else 'weak')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `signals = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score e-e-a-t signals from a dict and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Each present trust signal (author bio, credentials, enough citations, dates, about/contact pages) adds a point; the total maps to a strong/moderate/weak grade.

```
signals = {
    'author_bio': True,
    'author_credentials': True,
    'outbound_citations': 4,
```

```
    'published_date': True,
    'updated_date': False,
    'about_page': True,
    'contact_page': False,
}
score = 0
score += 1 if signals['author_bio'] else 0
score += 1 if signals['author_credentials'] else 0
score += 1 if signals['outbound_citations'] >= 2 else 0
score += 1 if signals['published_date'] else 0
score += 1 if signals['updated_date'] else 0
score += 1 if signals['about_page'] else 0
score += 1 if signals['contact_page'] else 0
print(f'E-E-A-T score: {score}/7')
print('grade:', 'strong' if score >= 6 else 'moderate' if score >= 4 else
'weak')
```

```
output
E-E-A-T score: 5/7
grade: moderate
```

How it works:

1. A signals dictionary records seven boolean or numeric trust cues for a single page.
2. Each line adds one to score when its signal is truthy, using a threshold for outbound citations.
3. The citation rule awards a point only when the count is at least two, not for any single link.
4. The total is printed as a fraction out of seven so the ceiling is always visible.
5. A chained conditional maps the score to strong, moderate or weak using six-and-four cutoffs.

Try it:

1. Weight credentials and citations more heavily than the about and contact page signals.
2. Replace the manual additions with a loop that sums booleans for cleaner, shorter code.
3. Add an HTTPS and a reviewed-by-expert signal to extend the checklist beyond seven points.
4. Raise the citation threshold to 3 and watch the score drop.

⚠ Common Mistakes

- Treating `outbound_citations` as a boolean ignores that 1 citation is weak; a `>= 2` threshold is used here.
- A missing dict key would raise `KeyError`; every signal must be present in the fixture.
- Counting raw `True` values without thresholds would over-credit a single citation.

Skills gained dict scoring | threshold checks | grade bucketing | E-E-A-T

→ Where You'll Use This: Quantifying trust signals lets agencies show clients a before-and-after E-E-A-T number, which is increasingly tied to whether AI engines will cite a page as a reliable source.

Mental Model

Like a credit score for a page: many small verifiable facts roll up into one number lenders, or engines, trust at a glance.

Project 472 Flag Stale Content Against an As-Of Date

The Idea Behind It

Freshness is both a ranking and a trust signal, and answer engines lean toward pages that were updated recently. The code parses a fixed audit date and each page's last-updated date with strptime, subtracts them to get an age in days, and flags any page older than a year as stale. Anchoring everything to one explicit AS_OF constant rather than today's clock makes the audit deterministic, so re-running it next month never silently changes which pages were considered stale at audit time.

What You'll Build & Learn

Flag pages whose last-updated date is more than a year before a fixed audit date as stale content.

- Parse dates with strptime
- Compute an age via date subtraction
- Anchor audits to a fixed as-of date

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 471: score e-e-a-t signals from a dict. Project 473 changes the surface problem while preserving the underlying method.

progress check

Reading target: flag stale content against an as-of date is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `from datetime import datetime` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`from datetime import datetime`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}')`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Flag Stale Content Against an As-Of Date: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: datetime, pages, guide, asof, strptime. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from datetime import datetime | Core: print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}') | Evidence: print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}')

🛠️ Reader to Writer Bridge

1. Read: explain flag stale content against an as-of date in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 471.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 473 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from datetime import datetime` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of flag stale content against an as-of date and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

`datetime.strptime` parses the embedded date strings (no clock call), and subtracting from a fixed as-of date gives an age in days that a 365-day threshold judges.

```
from datetime import datetime
AS_OF = '2026-06-22'
pages = {'/guide-a': '2026-05-01', '/guide-b': '2024-11-15', '/guide-c': '2026-06-10'}
asof = datetime.strptime(AS_OF, '%Y-%m-%d')
for path in sorted(pages):
    updated = datetime.strptime(pages[path], '%Y-%m-%d')
    age_days = (asof - updated).days
    stale = age_days > 365
    print(f'{path}: {age_days}d old -> {"STALE" if stale else "fresh"}')
```

```
output
/guide-a: 52d old -> fresh
/guide-b: 584d old -> STALE
/guide-c: 12d old -> fresh
```

How it works:

1. A constant AS_OF string and a dictionary of three page paths with update dates are defined.
2. strptime turns the as-of string into a datetime object the page dates can be compared against.
3. Iterating sorted paths, each page date is parsed and subtracted to yield an integer day age.
4. The stale flag is True whenever that age exceeds three hundred sixty-five days.
5. Each line prints the path, its age in days, and a STALE or fresh label.

Try it:

1. Make the staleness window configurable so news pages use ninety days and evergreen pages use a year.
2. Sort the output by age so the stalest pages surface at the top of the report.
3. Read the page dates and as-of value from a CSV file rather than a hard-coded dictionary.
4. Move `AS_OF` forward a year and watch more pages turn STALE.

⚠ Common Mistakes

- Using `datetime.now()` would be nondeterministic; a fixed `AS_OF` constant keeps the audit reproducible.
- Comparing date strings lexically works for ISO dates but breaks the day-count maths.
- An off-by-one in the threshold (`>=` vs `>`) reclassifies a page exactly one year old.

Skills gained date parsing | datetime math | freshness audit | deterministic runs

→ Where You'll Use This: Editorial and content-decay audits use exactly this logic to build refresh queues, prioritising the oldest evergreen pages that have drifted past a freshness threshold.

Mental Model

Treat content like food in a fridge: each page carries a date, and anything past its window gets pulled and refreshed before it spoils trust.

Project 473 Count Outbound Citations to Sources

The Idea Behind It

Linking out to authoritative sources is a trust signal both reviewers and answer engines weigh, so it pays to separate genuine citations from internal navigation. The code extracts every href with a regex, then partitions them into outbound links that start with https and internal links that start with a slash. Pulling the host out of each outbound URL with a split on the third path segment also reports which domains the page actually cites, turning a raw link count into a meaningful sourcing profile.

What You'll Build & Learn

Count a page's outbound citations versus internal links and list the external domains it references.

- Extract all hrefs with one regex
- Partition links by URL prefix
- Derive a host from a URL split

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 472: flag stale content against an as-of date. Project 474 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count outbound citations to sources is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('cited domains:', sorted({h.split('/')[2] for h in outbound}))` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`outbound = [h for h in hrefs if h.startswith('https://')]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print('cited domains:', sorted({h.split('/')[2] for h in outbound}))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `outbound = [h for h in hrefs if h.startswith('https://')]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Outbound Citations to Sources: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, Per, the, href. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: outbound = [h for h in hrefs if h.startswith('https://')] | Evidence: print('cited domains:', sorted({h.split('/')[2] for h in outbound}))

🛠️ Reader to Writer Bridge

1. Read: explain count outbound citations to sources in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 472.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 474 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('cited domains:', sorted({h.split('/')[2] for h in outbound}))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count outbound citations to sources and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Splitting hrefs by whether they start with `https://` (external) or `/` (internal) counts citations, and the host slice gives the distinct domains cited.

```
import re
html = '''<p>Per the <a href="https://standards.org/spec">spec</a>, see also
<a href="https://web.dev/lcp">web.dev</a> and <a href="/internal/page">our
guide</a>.</p>'''
hrefs = re.findall(r'href="([^"]+)"', html)
outbound = [h for h in hrefs if h.startswith('https://')]
internal = [h for h in hrefs if h.startswith('/')]
print('outbound citations:', len(outbound))
print('internal links:', len(internal))
print('cited domains:', sorted({h.split('/')[2] for h in outbound}))
```

```
output
outbound citations: 2
internal links: 1
cited domains: ['standards.org', 'web.dev']
```

How it works:

1. A multi-line HTML string contains two external https links and one internal slash-prefixed link.
2. re.findall captures every value inside an href attribute into a flat list of URLs.
3. A comprehension keeps links starting with https as outbound and slash-prefixed ones as internal.
4. Splitting each outbound URL on slashes and taking index two extracts the bare domain name.
5. Prints report the outbound count, the internal count, and the sorted set of cited domains.

Try it:

1. Use urllib.parse to extract hosts robustly instead of splitting on the third slash.
2. Flag nofollow links separately so sculpted and editorial citations are distinguished.
3. Treat links to the page's own domain as internal even when they use a full https URL.
4. Add another external link to the same domain and watch the citation count rise but domains stay.

⚠ Common Mistakes

- A relative link without a leading slash slips past the `/`-prefix internal test.
- Protocol-relative or uppercase-scheme URLs would not start with `https://` and be miscounted.
- `h.split('/')[2]` assumes a well-formed absolute URL; a malformed href would index out of range.

Skills gained href extraction | link partitioning | domain parsing | citation audit

→ Where You'll Use This: Assessing how well-sourced a page is, and which authorities it leans on, supports both E-E-A-T audits and competitive analysis of why rival pages earn more trust.

Mental Model

Citations are an academic paper's bibliography: a page that names credible sources reads as researched, while a sourceless one reads as opinion.

Project 474 Detect a Substantive Author Bio

The Idea Behind It

A named author with a real biography is a core trust cue, but a byline that is just a name proves little. This code confirms three things together: that an author-bio block exists in the markup, that a name can be captured from it, and that the bio paragraph carries at least ten words of substance. Requiring all three with a single combined boolean stops a decorative byline from passing, which is what separates genuine authorship signals from cosmetic ones.

What You'll Build & Learn

Verify a page has a real author block, a captured name, and a bio of meaningful length, not just a byline.

- Detect a block by class substring
- Capture nested text with re.S
- Gate on a minimum bio length

Skill Roadmap

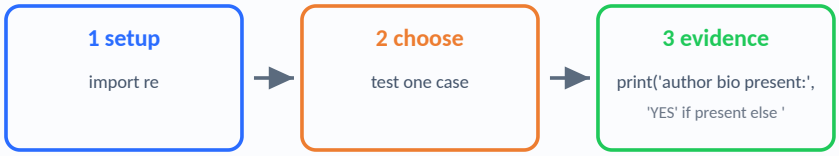
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 473: count outbound citations to sources. Project 475 changes the surface problem while preserving the underlying method. *progress check*

Reading target: detect a substantive author bio is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is about responsibility: data and behaviour are grouped under a name, so read methods as actions that belong to that object.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('author bio present:', 'YES' if present else 'NO')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`bio_words = len(bio_m.group(1).split()) if bio_m else 0`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. slow line
3. Check the evidence	`print('author bio present:', 'YES' if present else 'NO')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `bio_words = len(bio_m.group(1).split()) if bio_m else 0`

Construction pattern

The construction groups state and behavior under names. Read the class/record/interface header, then ask what each method is responsible for.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

List the fields or attributes first, then read methods as behavior attached to that stored state.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Substantive Author Bio: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, article, Body, text. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; classes/records group responsibilities; conditions choose a route.
Data shape	Object data: state is bundled with the methods that operate on it.
Evidence	Entry: import re | Core: bio_words = len(bio_m.group(1).split()) if bio_m else 0 | Evidence: print('author bio present:', 'YES' if present else 'NO')

🛠️ Reader to Writer Bridge

1. Read: explain detect a substantive author bio in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 473.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 475 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('author bio present:', 'YES' if present else 'NO')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of detect a substantive author bio and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Requiring an `author-bio` block, a captured name, and a bio of at least 10 words rejects empty or token bylines while accepting a genuine one.

```
import re
html = '''<article><p>Body text here.</p>
<div class="author-bio"><span class="author-name">Dana Lee</span>
<p>Dana is a senior SEO engineer with 10 years auditing technical sites.</p>
</div></article>'''
```

```
has_block = 'author-bio' in html
name_m = re.search(r'author-name">([^<]+)<', html)
bio_m = re.search(r'author-bio".*?<p>([^<]+)</p>', html, re.S)
bio_words = len(bio_m.group(1).split()) if bio_m else 0
present = has_block and name_m and bio_words >= 10
print('author block:', has_block)
print('author name:', name_m.group(1) if name_m else None)
print('bio words:', bio_words)
print('author bio present:', 'YES' if present else 'NO')
```

```
output
author block: True
author name: Dana Lee
bio words: 12
author bio present: YES
```

How it works:

1. An article string nests an author-bio div containing an author-name span and a biography paragraph.
2. A simple substring test sets has_block True when author-bio appears in the markup.
3. One regex captures the author name and another, using re.S, captures the bio paragraph across newlines.
4. The bio word count comes from splitting that captured paragraph, defaulting to zero if absent.
5. present is True only when the block, a name, and a ten-word bio all hold, and a YES prints.

Try it:

1. Also require a link to an author page or a sameAs profile to strengthen the signal.
2. Detect multiple author blocks on a page and report each name and bio length.
3. Raise the bio threshold and check that the bio mentions a relevant credential keyword.
4. Shorten the bio under 10 words and watch the verdict flip to NO.

⚠ Common Mistakes

- The bio regex uses `re.S` so `.*?` can cross newlines between the block and its `<p>`.
- A byline with a name but no descriptive bio passes the name test yet should fail on word count.
- Hard-coding the `author-bio` class name misses sites that mark up bios differently.

Skills gained substring detection | multiline regex | length gating | authorship signals

→ Where You'll Use This: Confirming bylines are backed by substantive author bios is central to E-E-A-T audits, especially for your-money-your-life pages where Google scrutinises authorship hardest.

⚡ Mental Model

A byline without a bio is a signature on a blank cheque: the name means nothing until something behind it vouches for the person.

Project 475 Count Distinct Entity Mentions

💡 The Idea Behind It

Knowledge-graph-aware engines read topical depth partly through entity coverage, the named tools, brands and concepts a page discusses. This project counts how many times each entity from a known list appears in a passage using an escaped regex inside a Counter, then reports the number of distinct entities mentioned at least once. Counting both per-entity frequency and distinct coverage matters because depth comes from breadth of entities, not just repeating one name many times.

What You'll Build & Learn

Count mentions of known entities in a text and report how many distinct ones the page covers.

- Count substring hits with re.escape
- Tally frequencies in a Counter
- Measure distinct entity coverage

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 474: detect a substantive author bio. Project 476 changes the surface problem while preserving the underlying method.

progress check

Reading target: count distinct entity mentions is the surface task; the transferable Python move is to mark setup, rule, and output before running.

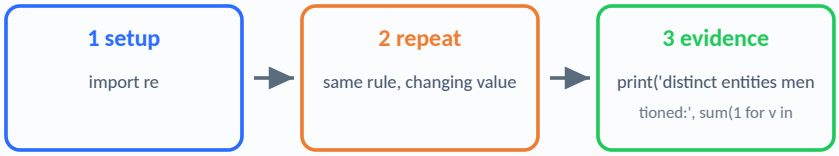
setup -> rule -> result

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The code is a small AI-engineering pipeline: prepare a value or dataset, apply a model/math step, then inspect a measurable result. Read it as `data -> transformation -> evidence`, not as isolated library calls.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.

- Evidence: `print('distinct entities mentioned:', sum(1 for v in counts.values() if v))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print('distinct entities mentioned:', sum(1 for v in counts.values() if v))`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*
3. Check the evidence	`print('distinct entities mentioned:', sum(1 for v in counts.values() if v))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('distinct entities mentioned:', sum(1 for v in counts.values() if v))`

Construction pattern

The construction is pipeline-shaped: data preparation -> model or transformation -> metric, prediction, or inspection output.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Separate the model step from the data step: one prepares evidence, the other learns, predicts, or scores.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Count Distinct Entity Mentions: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, collections, Counter, text, Google. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import re | Core: print('distinct entities mentioned:', sum(1 for v in counts.values() if... | Evidence: print('distinct entities mentioned:', sum(1 for v in counts.values() if...

🛠️ Reader to Writer Bridge

1. Read: explain count distinct entity mentions in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 474.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 476 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('distinct entities mentioned:', sum(1 for v in counts.values() if v))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of count distinct entity mentions and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Counting each entity phrase with `re.escape` tallies mentions, and summing the entities with a nonzero count gives breadth of entity coverage.

```
import re
from collections import Counter
text = '''Google Search Console reports indexing status. Search Console
connects to
Googlebot data, while Bing Webmaster Tools offers similar coverage for Bing.'''
entities = ['Google Search Console', 'Googlebot', 'Bing Webmaster Tools',
'Bing']
counts = Counter()
for e in entities:
    counts[e] = len(re.findall(re.escape(e), text))
for e in sorted(counts):
    print(f'{e}: {counts[e]}')
print('distinct entities mentioned:', sum(1 for v in counts.values() if v))
```

```
output
Bing: 2
Bing Webmaster Tools: 1
Google Search Console: 1
Googlebot: 1
distinct entities mentioned: 4
```

How it works:

1. A multi-line text mentions several SEO tools, and a list names the four entities to track.
2. A Counter is filled by running re.findall with re.escape for each entity against the text.
3. Escaping the entity protects literal characters so brand names match exactly, not as patterns.
4. Iterating the sorted Counter prints each entity alongside its raw mention count.
5. A final sum counts entities with a non-zero tally to report distinct coverage.

Try it:

1. Make matching case-insensitive and word-bounded so partial overlaps do not double-count.
2. Build the entity list automatically from a competitor's page rather than hard-coding it.
3. Report the share of expected entities covered as a percentage to grade topical completeness.
4. Use word-boundary matching to stop `Bing` counting inside the longer phrase.

⚠ Common Mistakes

- `Bing` shows 2 because it is also a substring of `Bing Webmaster Tools`; overlapping entities double-count.
- Without `re.escape`, an entity containing regex metacharacters would break the pattern.
- Case-sensitive matching misses `bing` written in lower case in the body.

Skills gained `entity counting` `Counter usage` `regex escaping` `topical depth`

→ **Where You'll Use This:** Entity-coverage analysis informs content briefs, showing which related tools, brands or concepts a page must mention to read as authoritative to entity-driven AI search.

⚡ Mental Model

Entities are the cast list of a topic; a thin page names one or two actors, while a deep one introduces the whole ensemble.

Project 476 Classify a Block's Answer Format

💡 The Idea Behind It

Answer engines extract differently from lists, tables, definitions and plain prose, so labelling each block by its format predicts how it might be reused. This code walks a dictionary of HTML blocks and applies an ordered set of tests: list tags first, then table tags, then a definition pattern matching an X is a opener, falling through to prose. The ordering is deliberate, because the most distinctive structural markers are checked before the looser linguistic one, avoiding misclassification.

What You'll Build & Learn

Classify each content block as a list, table, definition, or plain prose by its markup and phrasing.

- Order tests from specific to general
- Match a definition with a regex opener
- Map blocks to answer-format labels

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 475: count distinct entity mentions. Project 477 changes the surface problem while preserving the underlying method.

progress check

Reading target: classify a block's answer format is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

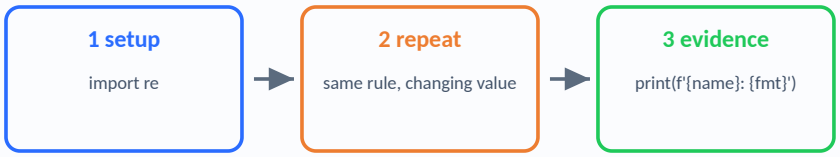
🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a

32-point technical audit.

- Evidence: `print(f'{name}: {fmt}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if re.search(r'<ol>|<ul>', h):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{name}: {fmt}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if re.search(r'<ol>|<ul>', h):`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Read the try path as the hoped-for route and the catch/except/finally path as the boundary plan.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Classify a Block's Answer Format: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, blocks, definition, Crawl, budget. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: if re.search(r' | ', h): | Evidence: print(f'{name}: {fmt}')

🛠️ Reader to Writer Bridge

1. Read: explain classify a block's answer format in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 475.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 477 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{name}: {fmt}')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classify a block's answer format and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Ordered checks (list, then table, then a `term is/are a ...` definition pattern, else prose) bucket each block into the format an engine would most likely lift.

```
import re
blocks = {
    'definition': '<p>Crawl budget is the number of pages Googlebot will fetch.
</p>',
    'steps': '<ol><li>Open Search Console</li><li>Submit sitemap</li></ol>',
    'comparison': '<table><tr><th>301</th><th>302</th></tr></table>',
    'plain': '<p>We think SEO is great and you should try it sometime soon.
</p>',
}
for name, h in blocks.items():
    if re.search(r'<ol>|<ul>', h):
        fmt = 'list/steps'
    elif '<table' in h:
        fmt = 'table/comparison'
    elif re.match(r'<p>\w+( \w+)? (is|are) (a |an |the )', h):
        fmt = 'definition'
    else:
        fmt = 'prose'
    print(f'{name}: {fmt}')
```

```
output
definition: definition
steps: list/steps
comparison: table/comparison
plain: prose
```

How it works:

1. A dictionary pairs four named blocks with HTML for a definition, steps, a comparison, and plain text.
2. For each block, an ordered chain tests for list tags, then a table tag, then a definition pattern.
3. The definition regex matches a paragraph opening with an optional article and an is or are verb.
4. Any block matching none of those rules falls through to the prose label.
5. Each block name prints next to its inferred answer-format category.

Try it:

1. Add a steps-versus-unordered-list distinction so how-to content is labelled separately.
2. Return a confidence score alongside each label rather than a single hard category.
3. Handle blocks that mix formats, such as a paragraph followed by a list, with a combined tag.
4. Add a numbered `<ol>` block and confirm it reads as list/steps.

⚠ Common Mistakes

- The definition pattern requires an article after `is/are`, so `SEO is great` correctly falls through to prose.
- Checking definition before lists would misclassify a list item that happens to read like a sentence.

- A `<table>` used for layout still classifies as a comparison table here.

Skills gained `block classification` `ordered conditionals` `definition regex` `format mapping`

→ **Where You'll Use This:** Profiling the mix of answer formats across a page or site shows whether content is shaped for extraction, guiding where to convert dense prose into lists or tables.

⚡ Mental Model

It is a recycling sorter: each block drops down a chute of shape tests until it lands in the bin labelled list, table, definition or prose.

Project 477 Run an Answer-Box Readiness Checklist

💡 The Idea Behind It

Winning an answer box needs several conditions at once, so this project scores readiness against a five-item checklist rather than any single test. It stores five boolean page properties, pairs each with a human-readable label in a list of tuples, then sums the passing ones and prints a checkbox line for each. The tuple-of-label-and-flag structure is the useful pattern: it keeps the display text beside the data, so the report and the score are driven by one source of truth.

What You'll Build & Learn

Score a page against a five-point answer-box checklist and render a readable pass-or-fail report.

- Pair labels with flags in tuples
- Sum booleans to score a checklist
- Render checkbox-style report lines

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 476: classify a block's answer format. Project 478 changes the surface problem while preserving the underlying method.

progress check

Reading target: run an answer-box readiness checklist is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

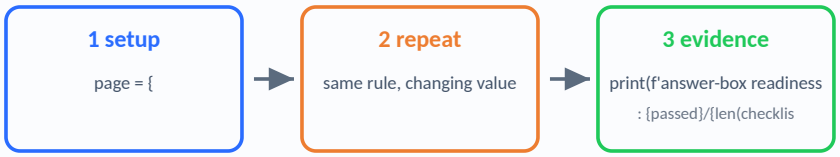
🔎 Logic Lens

- Setup: `page = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'answer-box readiness: {passed}/{len(checklist)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`page = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`passed = sum(1 for _, ok in checklist if ok)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'answer-box readiness: {passed}/{len(checklist)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `passed = sum(1 for _, ok in checklist if ok)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run an Answer-Box Readiness Checklist: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: page, question_heading, concise_answer_first, answer_under_60_words, supporting_detail. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: page = { | Core: passed = sum(1 for _, ok in checklist if ok) | Evidence: print(f'answer-box readiness: {passed}/{len(checklist)}')

🛠️ Reader to Writer Bridge

1. Read: explain run an answer-box readiness checklist in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 476.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 478 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'answer-box readiness: {passed}/{len(checklist)}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `page = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run an answer-box readiness checklist and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Each readiness criterion is a boolean in the page dict; printing a ticked checklist and the pass tally turns the abstract goal into an actionable score.

```
page = {
    'question_heading': True,
    'concise_answer_first': True,
    'answer_under_60_words': True,
    'supporting_detail': True,
    'schema_qapage': False,
}
checklist = [
    ('Question-style heading', page['question_heading']),
    ('Concise answer up top', page['concise_answer_first']),
    ('Answer under 60 words', page['answer_under_60_words']),
    ('Supporting detail below', page['supporting_detail']),
    ('QAPage schema', page['schema_qapage']),
]
passed = sum(1 for _, ok in checklist if ok)
for name, ok in checklist:
    print(f'[{"x" if ok else " "}] {name}')
print(f'answer-box readiness: {passed}/{len(checklist)}')
```

```
output
[x] Question-style heading
[x] Concise answer up top
[x] Answer under 60 words
[x] Supporting detail below
[ ] QAPage schema
answer-box readiness: 4/5
```

How it works:

1. A page dictionary records five answer-box readiness booleans, one of them False for QAPage schema.
2. A checklist list pairs each human-readable criterion name with its corresponding page flag.
3. passed sums the truthy flags by iterating the tuples and counting where ok is True.
4. A loop prints each criterion with an x in brackets when passed and a space when not.
5. The closing line reports the score as passed over the total number of checklist items.

Try it:

1. Append the failing criteria to a separate list and print them as next-step actions.
2. Add weights so the concise-answer item counts more than the supporting-detail one.
3. Read the page flags from a real HTML parse instead of a pre-filled dictionary.
4. Flip `schema_qapage` to True and watch the score hit 5/5.

⚠ Common Mistakes

- Reading missing dict keys would raise; every criterion must be present in the fixture.
- Counting `True` values without printing them hides which criterion failed.
- Treating the 5/5 as required ignores that 4/5 can still win an answer box.

Skills gained checklist scoring | tuple pairing | boolean summing | answer-box readiness

→ Where You'll Use This: This checklist-plus-report pattern is the backbone of client-facing audit deliverables, where each unchecked item becomes a concrete recommendation to chase an answer box.

Mental Model

Like a pre-flight checklist: every box must be ticked before the plane, or the page, is cleared to claim the answer box.

Project 478 Score AI-Mode Readiness With Weights

The Idea Behind It

AI-mode search rewards clear structure, schema, concise answers, freshness and citations, but it does not value them equally, so a weighted score is more honest than a flat count. The code holds page signals in one dictionary and per-signal weights in another, sums the weights of the truthy signals, and converts that to a percentage of the maximum. Looking up each signal's weight only when its flag is True is the crux: it lets the most important factors, like concise answers, dominate the readiness number.

What You'll Build & Learn

Score AI-mode readiness with per-signal weights, give a percentage verdict, and list what is missing.

- Weight signals with a parallel dict
- Sum weights only for passing signals
- Convert a score into a percentage

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 477: run an answer-box readiness checklist. Project 479 changes the surface problem while preserving the underlying method.

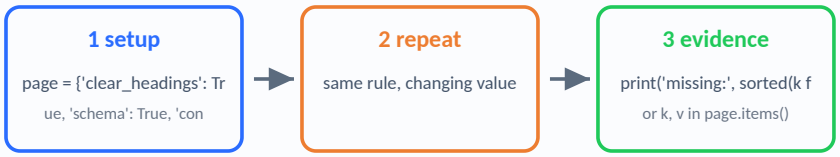
progress check

Reading target: score ai-mode readiness with weights is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `page = {'clear_headings': True, 'schema': True, 'concise_answers': True, 'fresh': False, 'cited_sources': True}` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('missing:', sorted(k for k, v in page.items() if not v))` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`page = {'clear_headings': True, 'schema': True, 'concise_answers': True, 'fresh': False, 'cited_sources': True}`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`got = sum(weights[k] for k, v in page.items() if v)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('missing:', sorted(k for k, v in page.items() if not v))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `got = sum(weights[k] for k, v in page.items() if v)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Score AI-Mode Readiness With Weights: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: page, clear_headings, schema, concise_answers, fresh. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: page = {'clear_headings': True, 'schema': True, 'concise_answers': True... | Core: got = sum(weights[k] for k, v in page.items() if v) | Evidence: print('missing:', sorted(k for k, v in page.items() if not v))

🛠️ Reader to Writer Bridge

1. Read: explain score ai-mode readiness with weights in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 477.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 479 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('missing:', sorted(k for k, v in page.items() if not v))`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `page = {'clear_headings': True, 'schema': True, 'concise_answers': True, 'fresh': False, 'cited_sources': True}` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of score ai-mode readiness with weights and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Summing the weights of only the present signals over the total weight gives a percentage; concise answers and schema carry more weight than freshness.

```
page = {'clear_headings': True, 'schema': True, 'concise_answers': True,
'fresh': False, 'cited_sources': True}
weights = {'clear_headings': 2, 'schema': 2, 'concise_answers': 3, 'fresh': 1,
'cited_sources': 2}
got = sum(weights[k] for k, v in page.items() if v)
total = sum(weights.values())
pct = round(100 * got / total)
print(f'AI-mode readiness: {got}/{total} ({pct}%)')
print('verdict:', 'ready' if pct >= 70 else 'needs work')
print('missing:', sorted(k for k, v in page.items() if not v))
```

```
output
AI-mode readiness: 9/10 (90%)
verdict: ready
missing: ['fresh']
```

How it works:

1. One dictionary stores five readiness booleans and a parallel dictionary stores each signal's weight.
2. got sums the weight of every signal whose flag is True by iterating the page items.
3. total sums all weights so the percentage uses the full possible score as its denominator.
4. The percentage is rounded and compared against seventy to decide ready or needs work.
5. A final print lists the sorted names of any signals that are still False.

Try it:

1. Tune the weights from observed correlation with citations rather than fixed guesses.
2. Add an interpolated grade band so scores near the threshold are not a binary cliff.
3. Print each missing signal alongside the points it would add if fixed.
4. Bump the `fresh` weight to 3 and watch the percentage fall.

⚠ Common Mistakes

- Using equal weights would over-value freshness relative to concise answers.
- Dividing by the count of present signals instead of total weight inflates the percentage.
- A signal absent from the weights dict would raise `KeyError` during the sum.

Skills gained weighted scoring | percentage math | verdict thresholds | AI-mode readiness

→ Where You'll Use This: A weighted readiness score lets you prioritise fixes by impact when preparing pages for generative search, focusing effort on the heavily weighted signals first.

Mental Model

Like a recipe where some ingredients are essential and others optional: leaving out the salt costs more than skipping a garnish.

Project 479 Count Extractable Lists and Tables

The Idea Behind It

Lists and tables are the structures answer engines most readily lift into rich results, so counting them gauges how much extractable structure a page offers. The code uses regexes to find unordered-list blocks, individual list items, and table rows, then computes a simple extractable-structures total that adds the list count plus one if any table rows exist. Treating an entire table as a single extractable unit while counting lists individually reflects how engines reuse a whole table versus separate list blocks.

What You'll Build & Learn

Count the lists, list items, and table rows on a page and report how many extractable structures exist.

- Match block and item tags separately
- Count table rows with a row regex
- Combine counts into one structure total

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 478: score ai-mode readiness with weights. Project 480 changes the surface problem while preserving the underlying method.

progress check

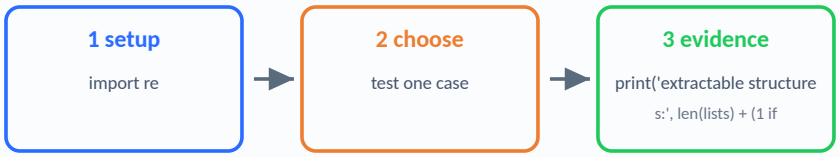
Reading target: count extractable lists and tables is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('extractable structures:', len(lists) + (1 if rows else 0))` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print('extractable structures:', len(lists) + (1 if rows else 0))`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('extractable structures:', len(lists) + (1 if rows else 0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('extractable structures:', len(lists) + (1 if rows else 0))`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Count Extractable Lists and Tables: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, h2, Sitemap, Limits. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print('extractable structures:', len(lists) + (1 if rows else 0)) | Evidence: print('extractable structures:', len(lists) + (1 if rows else 0))

🛠️ Reader to Writer Bridge

1. Read: explain count extractable lists and tables in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 478.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 480 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('extractable structures:', len(lists) + (1 if rows else 0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of count extractable lists and tables and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Regexes over `<ul>`/`<li>`/`<tr>` count the list blocks, their items, and the table rows; a list plus a table is two extractable structures.

```
import re
html = '''<h2>Sitemap Limits</h2>
<ul><li>50,000 URLs max</li><li>50MB uncompressed</li><li>One sitemap
index</li></ul>
<table><tr><th>Type</th><th>Limit</th></tr><tr><td>URLs</td><td>50000</td></tr>
</table>'''
lists = re.findall(r'<ul>(.*?)</ul>', html, re.S)
list_items = re.findall(r'<li>(.*?)</li>', html)
rows = re.findall(r'<tr>', html)
print('lists found:', len(lists))
print('list items:', len(list_items))
print('table rows:', len(rows))
print('extractable structures:', len(lists) + (1 if rows else 0))
```

```
output
lists found: 1
list items: 3
table rows: 2
extractable structures: 2
```

How it works:

1. A multi-line HTML string holds a heading, a three-item unordered list, and a small two-row table.
2. re.findall with re.S captures whole list blocks, while a second call counts individual list items.
3. A third findall counts opening table-row tags to measure the table's size.
4. The extractable total adds the number of lists plus one when any table rows were found.
5. Four prints report the list count, item count, row count, and combined extractable-structure total.

Try it:

1. Include ordered lists and definition lists so every structured block is counted.
2. Count each table as a structure rather than collapsing all tables into a single unit.
3. Report items per list so a one-item list does not count the same as a rich ten-item list.
4. Add a second `<ul>` and watch the list count rise.

⚠ Common Mistakes

- A greedy `<ul>(.*)</ul>` across two lists would merge them into one match; `.*?` with `re.S` keeps them separate.
- Counting `<tr>` includes the header row, so subtract one for data-row counts if needed.
- Nested lists are counted as separate `<ul>` blocks, which can overstate distinct structures.

Skills gained structure counting | regex with re.S | row tallying | extractability

→ Where You'll Use This: Quantifying extractable structures helps prioritise which dense pages to restructure into lists and tables when chasing rich results and AI-citation eligibility.

⚡ Mental Model

Lists and tables are pre-cut handles on the page; the more handles, the easier it is for an engine to grab a piece and carry it away.

Project 480 Detect Definition-Format Sentences

💡 The Idea Behind It

A clean definition sentence is exactly what answer engines quote for what-is queries, so detecting that pattern surfaces a page's quotable lines. The code compiles a case-insensitive regex that matches a sentence opening with an optional article, a captured subject word, and a defining verb such as is, are, means or refers to. Running match rather than search anchors the test to the start of each paragraph, which keeps it from firing on a defining phrase buried mid-sentence where it would not read as a clean definition.

What You'll Build & Learn

Detect paragraphs written as clean definitions and report the term each one defines.

- Compile a reusable regex pattern
- Anchor a match to the sentence start
- Capture the defined subject term

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 479: count extractable lists and tables. Project 481 changes the surface problem while preserving the underlying method.

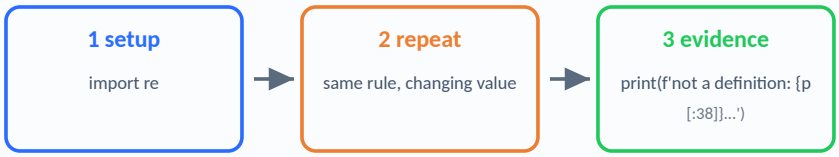
progress check

Reading target: detect definition-format sentences is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'not a definition: {p[:38]}...')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if m:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'not a definition: {p[:38]}...')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if m:`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Definition-Format Sentences: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, paras, Canonicalization, is, the. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: if m: | Evidence: print(f'not a definition: {p[:38]}...')

🛠️ Reader to Writer Bridge

1. Read: explain detect definition-format sentences in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 479.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 481 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'not a definition: {p[:38]}...')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of detect definition-format sentences and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

An anchored regex allowing an optional leading article then `term is/are/means/refers to` captures the defined term and rejects sentences that merely contain `is`.

```
import re
paras = [
    'Canonicalization is the process of selecting one URL among duplicates.',
    'You should always check your robots file before launch.',
    'A redirect is a server response that forwards one URL to another.',
]
pat = re.compile(r'^(?:A |An |The )?(\w+) (?:is|are|means|refers to) ', re.I)
for p in paras:
    m = pat.match(p)
    if m:
        print(f'DEFINITION of {m.group(1)!r}')
    else:
        print(f'not a definition: {p[:38]}...')
```

```
output
DEFINITION of 'Canonicalization'
not a definition: You should always check your robots fi...
DEFINITION of 'redirect'
```

How it works:

1. A list holds three paragraphs, two of which are phrased as definitions and one as advice.
2. re.compile builds a case-insensitive pattern with an optional leading article and a defining verb.
3. Using match, the pattern only succeeds when the paragraph starts with the definition shape.
4. When it matches, group one yields the subject term, printed as DEFINITION of that term.
5. Non-matching paragraphs print a truncated preview labelled not a definition.

Try it:

1. Capture multi-word subjects so a phrase like canonical URL is reported whole.
2. Add more defining verbs such as describes and denotes to widen detection.
3. Score each definition by length and flag ones too long to be quoted cleanly.
4. Add `consists of` to the verb alternation and test a new sentence.

⚠ Common Mistakes

- Requiring the verb to be followed by a space stops `is.` style false ends, but a comma after the term would still pass.
- An imperative sentence like `You should check ...` has no defining verb early, so it falls through.
- Multi-word terms before the verb are collapsed to one captured token here.

Skills gained compiled regex | anchored matching | group capture | definition detection

→ Where You'll Use This: Finding a page's definition sentences pinpoints the lines most likely to be quoted in featured snippets and AI answers, and shows where a missing definition could be added.

⚡ Mental Model

Think of a dictionary entry: the headword followed by a crisp gloss is exactly the shape an engine reaches for when someone asks what something means.

Project 481 Detect a Comparison Table

💡 The Idea Behind It

Comparison tables answer X versus Y queries and earn table-style rich results, so it helps to tell a true comparison from an incidental table. The code captures the header row, extracts its column headers, and separates the body rows, then declares a comparison only when there are three or more columns and at least one body row. Requiring three columns is the insight: a feature column plus two or more compared items is what makes a table a side-by-side comparison rather than a simple two-column list.

What You'll Build & Learn

Detect whether a table is a genuine comparison by checking its columns and body rows.

- Capture a table's header row
- Extract column headers with regex
- Define comparison by column count

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 480: detect definition-format sentences. Project 482 changes the surface problem while preserving the underlying method.

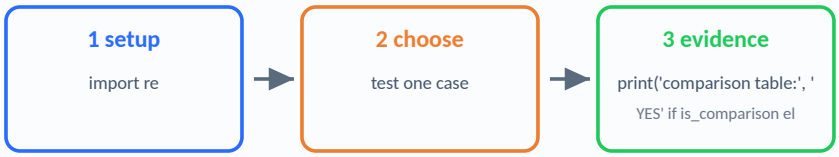
progress check

Reading target: detect a comparison table is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The condition splits the world into cases. Read the test aloud, then follow only the branch that matches the current data.
- Writing move: this project practises how to express a choice clearly enough that only one branch wins. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('comparison table:', 'YES' if is_comparison else 'NO')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print('compared items:', cols[1:] if cols else [])`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('comparison table:', 'YES' if is_comparison else 'NO')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('compared items:', cols[1:] if cols else [])`

Construction pattern

The construction splits into cases. Translate the condition into plain English, then follow only the matching branch.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Make a branch table with condition true/false and the exact statement that runs in each case.

Rewrite drill

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Detect a Comparison Table: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, html, table, tr, th. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import re | Core: print('compared items:', cols[1:] if cols else []) | Evidence: print('comparison table:', 'YES' if is_comparison else 'NO')

🛠️ Reader to Writer Bridge

1. Read: explain detect a comparison table in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 480.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 482 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('comparison table:', 'YES' if is_comparison else 'NO')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of detect a comparison table and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The first `<tr>` holds the header; three or more `<th>` columns and at least one body `<tr>` indicate a comparison of the items named after the leading label column.

```
import re
html = '''<table>
<tr><th>Feature</th><th>301</th><th>302</th></tr>
<tr><td>Permanent</td><td>Yes</td><td>No</td></tr>
<tr><td>Passes equity</td><td>Yes</td><td>Partial</td></tr>
</table>'''
header = re.search(r'<tr>(.*?)</tr>', html, re.S).group(1)
cols = re.findall(r'<th>(.*?)</th>', header)
body_rows = re.findall(r'<tr>(.*?)</tr>', html, re.S)[1:]
is_comparison = len(cols) >= 3 and len(body_rows) >= 1
print('columns:', cols)
print('compared items:', cols[1:] if cols else [])
print('comparison table:', 'YES' if is_comparison else 'NO')
```

```
output
columns: ['Feature', '301', '302']
compared items: ['301', '302']
comparison table: YES
```

How it works:

1. A multi-line HTML table compares two redirect types across two feature rows.
2. re.search with re.S captures the first row, and findall pulls its header cells into a list.
3. Slicing all rows from index one onward separates the body rows from the header.
4. is_comparison is True when there are at least three columns and one or more body rows.
5. Prints report the columns, the compared items after the first column, and the verdict.

Try it:

1. Validate that every body row has the same number of cells as the header.
2. Report which feature differs most between the compared items by scanning the body cells.
3. Handle tables whose header uses table-data cells instead of table-header cells.
4. Drop a column so only two remain and watch the verdict flip to NO.

⚠ Common Mistakes

- Using `re.S` is needed because the rows span newlines in the fixture.

- Counting `<th>` across the whole table, not just the header row, would overcount columns.
- A two-column table is a key/value list, not a comparison, so the `>= 3` test excludes it.

Skills gained table parsing | header extraction | row slicing | comparison detection

→ Where You'll Use This: Identifying real comparison tables helps target X-versus-Y queries, since these tables both serve users and earn the table rich results that drive clicks.

⚡ Mental Model

A comparison table is a head-to-head scorecard: one column names the criteria and each rival gets its own column to be judged against them.

Project 482 Detect Statistics Backed by a Source

💡 The Idea Behind It

A statistic with a named source is far more trustworthy and citable than a bare number, so this project separates cited stats from unsourced ones. It splits the text into sentences with a lookbehind regex, then for each sentence tests two independent conditions: whether a number with a percent marker appears, and whether a source cue like a parenthetical Source or according to is present. Printing only the sentences that contain a statistic, labelled by whether a source backs them, turns prose into a list of claims to keep or fix.

What You'll Build & Learn

Split text into sentences and flag each statistic as cited or unsourced based on a named source cue.

- Split sentences with a lookbehind
- Detect a percentage with regex
- Test for source-attribution cues

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 481: detect a comparison table. Project 483 changes the surface problem while preserving the underlying method.

progress check

Reading target: detect statistics backed by a source is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import re` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('UNSOURCED STAT:', s)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import re`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`sents = [s.strip() for s in re.split(r'(?<=[.!?])\s+', text) if s.strip()]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('UNSOURCED STAT:', s)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sents = [s.strip() for s in re.split(r'(?<=[.!?])\s+', text) if s.strip()]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Detect Statistics Backed by a Source: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: re, text, Organic, traffic, grew. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: import re | Core: sents = [s.strip() for s in re.split(r'(?<=[.!?])\s+', text) if s.strip... | Evidence: print('UNSOURCED STAT:', s)

🛠️ Reader to Writer Bridge

1. Read: explain detect statistics backed by a source in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 481.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 483 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('UNSOURCED STAT:', s)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import re` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of detect statistics backed by a source and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Splitting into sentences, then testing each for a number-with-percent and a source cue (`Source:`, `according to`, `per`) separates cited statistics from unsourced ones.

```
import re
text = '''Organic traffic grew 53% in 2025 (Source: BrightEdge).
Most pages load in under 2 seconds.
Pages with schema earn 30 percent more clicks according to Search Engine
Land.'''
sents = [s.strip() for s in re.split(r'(?<=[.!?])\s+', text) if s.strip()]
for s in sents:
    has_stat = bool(re.search(r'\d+\s?(%|percent)', s))
    has_source = bool(re.search(r'\(Source:|according to|per ', s, re.I))
    if has_stat and has_source:
        print('CITED STAT:', s)
    elif has_stat:
        print('UNSOURCED STAT:', s)
```

```
output
CITED STAT: Organic traffic grew 53% in 2025 (Source: BrightEdge).
CITED STAT: Pages with schema earn 30 percent more clicks according to Search Engine
Land.
```

How it works:

1. A multi-line text mixes two sourced statistics with one vague unsourced claim.
2. A regex split on sentence-ending punctuation with a lookbehind yields trimmed sentences.
3. Each sentence is tested for a digit followed by a percent sign or the word percent.
4. A second test looks for a Source parenthetical, the phrase according to, or a leading per.
5. Sentences with a stat print as CITED STAT when a source is found and UNSOURCED STAT otherwise.

Try it:

1. Extract and report the cited source name alongside each flagged statistic.
2. Detect plain numbers and currency figures, not only percentages, as statistics.
3. Count the ratio of cited to total stats as a page-level sourcing score.
4. Add a stat with no source and watch it print as UNSOURCED.

⚠ Common Mistakes

- The sentence split on `(?<=[.!?])\s+` keeps `2.5 seconds` intact only because it has no following space-capital; tune the lookbehind for decimals.
- A statistic written as `half of pages` has no digit and is missed entirely.
- `according to` inside an unrelated clause could falsely mark a non-statistic as cited.

Skills gained sentence splitting | stat detection | source attribution | claim auditing

→ Where You'll Use This: Flagging unsourced statistics is a fact-checking step in content audits, since AI engines and reviewers favour pages whose numbers carry verifiable attributions.

Mental Model

A statistic without a source is a rumour with a number attached; a citation is the receipt that lets a reader, or an engine, verify the claim.

Project 483 Write a robots.txt Audit Check

The Idea Behind It

The very first audit check has to confirm crawlers can reach and read the site at all, because nothing downstream matters if they cannot. This project defines check_robots, which safely pulls a nested robots dictionary, reads whether the file is reachable, and reads whether it disallows everything, returning a pass only when the file is reachable and not blocking the whole site. Using dict.get with defaults makes the function robust against missing keys, so an incomplete site fixture fails gracefully rather than raising.

What You'll Build & Learn

Write a robots audit check that passes only when the robots file is reachable and not blocking the whole site.

- Safely read nested dict values
- Combine two failure conditions
- Return a clear pass-or-fail boolean

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 482: detect statistics backed by a source. Project 484 changes the surface problem while preserving the underlying method.

progress check

Reading target: write a robots.txt audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running.

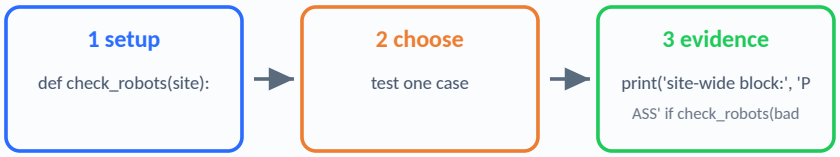
setup -> rule -> result

Logic Lens

- Setup: `def check_robots(site):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The function boundary is the lesson: inputs enter, a rule runs, and a return value can be reused somewhere else.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('site-wide block:', 'PASS' if check_robots(bad) else 'FAIL')` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, choose applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def check_robots(site):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print('robots.txt check:', 'PASS' if ok else 'FAIL')`	This line chooses a case. Read the test in plain English, then follow only the branch that matches. *slow line*
3. Check the evidence	`print('site-wide block:', 'PASS' if check_robots(bad) else 'FAIL')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print('robots.txt check:', 'PASS' if ok else 'FAIL')`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write a robots.txt Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: check_robots, site, get, robots, reachable. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def check_robots(site): | Core: print('robots.txt check:', 'PASS' if ok else 'FAIL') | Evidence: print('site-wide block:', 'PASS' if check_robots(bad) else 'FAIL')

🛠️ Reader to Writer Bridge

1. Read: explain write a robots.txt audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 482.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 484 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('site-wide block:', 'PASS' if check_robots(bad) else 'FAIL')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `def check_robots(site):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write a robots.txt audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

A tiny function over a site-signal dict returns pass only when robots.txt is reachable and does not carry a site-wide Disallow, the building block of the larger audit.

```
def check_robots(site):
    r = site.get('robots', {})
    reachable = r.get('reachable', False)
    blocks_all = r.get('disallow_all', False)
    return reachable and not blocks_all
site = {'robots': {'reachable': True, 'disallow_all': False}}
ok = check_robots(site)
print('robots.txt check:', 'PASS' if ok else 'FAIL')
bad = {'robots': {'reachable': True, 'disallow_all': True}}
print('site-wide block:', 'PASS' if check_robots(bad) else 'FAIL')
```

```
output
robots.txt check: PASS
site-wide block: FAIL
```

How it works:

1. check_robots reads a nested robots dictionary from a site, defaulting to empty when absent.
2. It pulls reachable and disallow_all flags with get so missing keys default to False.
3. The function returns True only when the file is reachable and does not disallow everything.
4. A healthy site fixture is checked and prints PASS for a reachable, open robots file.
5. A second fixture that disallows all is checked and prints FAIL for the site-wide block.

Try it:

1. Also fail when the robots file returns a non-200 status or times out.
2. Parse specific disallow rules and warn when key sections are blocked, not only all.
3. Confirm the robots file references a sitemap and report its absence.
4. Add a `crawl_delay` signal and fail values above 30 seconds.

⚠ Common Mistakes

- Treating a missing `robots` key as a pass would hide an unreachable file; `.get` defaults guard that.
- Confusing `disallow_all` (block everything) with a targeted Disallow over-fails normal sites.

- A reachable file that blocks all bots still fails despite being present.

Skills gained robots audit | safe dict access | boolean combination | crawlability

→ Where You'll Use This: A reusable robots check is the gateway of any automated technical audit, catching the catastrophic case where a stray disallow rule has deindexed an entire site.

Mental Model

Robots.txt is the front door's welcome mat: if the door is locked or the mat says keep out, no other room in the house gets seen.

Project 484 Write a Sitemap Audit Check

The Idea Behind It

A sitemap audit confirms a valid, in-bounds sitemap exists, since discovery quietly breaks without one. The check_sitemap function reads a nested sitemap dictionary and returns True only when it is present, holds a URL count strictly above zero and within the fifty-thousand limit, and reports valid XML. Chaining the present, the bounded count, and the validity tests with and in one expression means any single failing condition rejects the sitemap, which is how the healthy, missing and oversized cases each resolve correctly.

What You'll Build & Learn

Write a sitemap audit check that passes only for a present, valid, in-bounds sitemap and run it on three cases.

- Bound a count between two limits
- Chain conditions in one return
- Test multiple fixtures in a loop

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 483: write a robots.txt audit check. Project 485 changes the surface problem while preserving the underlying method.

progress check

Reading target: write a sitemap audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

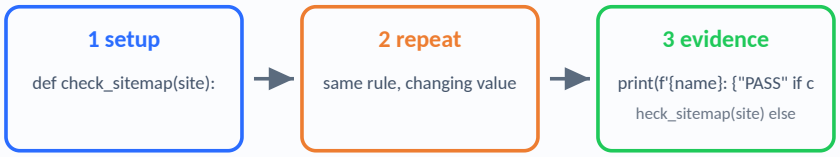
Logic Lens

- Setup: `def check_sitemap(site):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape.

repeat, test, update

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def check_sitemap(site):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}')`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write a Sitemap Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: check_sitemap, site, get, sitemap, bool. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def check_sitemap(site): | Core: print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}') | Evidence: print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}')

🛠️ Reader to Writer Bridge

1. Read: explain write a sitemap audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 483.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 485 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `def check_sitemap(site):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write a sitemap audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The check passes only when the sitemap is present, holds between 1 and 50,000 URLs, and parses as valid XML, mirroring the real sitemap limits.

```
def check_sitemap(site):
    s = site.get('sitemap', {})
    return bool(s.get('present')) and 0 < s.get('url_count', 0) <= 50000 and
s.get('valid_xml', False)
cases = {
    'healthy': {'sitemap': {'present': True, 'url_count': 1200, 'valid_xml':
True}},
    'missing': {'sitemap': {'present': False, 'url_count': 0, 'valid_xml':
False}},
    'oversized': {'sitemap': {'present': True, 'url_count': 60000, 'valid_xml':
True}},
}
for name, site in cases.items():
    print(f'{name}: {"PASS" if check_sitemap(site) else "FAIL"}')
```

```
output
healthy: PASS
missing: FAIL
oversized: FAIL
```

How it works:

1. check_sitemap reads a nested sitemap dictionary, defaulting to empty when the key is absent.
2. The return chains present, a URL count above zero and at or below fifty thousand, and valid XML.
3. A cases dictionary defines a healthy, a missing, and an oversized sitemap fixture.
4. Iterating the cases, each fixture is passed to the check function.
5. Each name prints PASS or FAIL, so only the healthy in-bounds sitemap passes.

Try it:

1. Add a check that a sitemap index is used once the URL count nears the per-file limit.
2. Validate the lastmod dates inside the sitemap are present and recent.
3. Fail when the sitemap is present but its declared count and actual entries disagree.
4. Raise the cap to 60,000 and watch the oversized case pass.

⚠ Common Mistakes

- A `url_count` of 0 must fail even if `present` is True; the `0 < count` bound enforces it.
- Forgetting the 50,000-URL cap lets an oversized sitemap pass.
- Skipping the `valid_xml` flag would accept a present-but-broken sitemap.

Skills gained sitemap audit | range bounding | condition chaining | fixture testing

→ Where You'll Use This: Validating sitemap presence, size and well-formedness is standard in crawl audits, since an oversized or malformed sitemap silently strands pages outside discovery.

Mental Model

A sitemap is the building directory in the lobby: missing, blank, or wildly over-length, it sends visitors and crawlers to the wrong floors.

Project 485 Write a Canonical Audit Check

The Idea Behind It

The canonical check confirms each page declares itself as its own canonical, because a missing or misdirected tag risks the page being dropped in favour of another URL. check_canonical reads the page's canonical and its own URL, returning a tuple of a pass flag plus a reason string: missing when no tag exists, points elsewhere when they differ, and self-referencing when they match. Returning the reason alongside the boolean is the design choice that lets the audit explain a failure, not merely report it.

What You'll Build & Learn

Write a canonical audit check that returns a pass flag and a reason for each page's canonical tag.

- Return a boolean plus a reason
- Detect a self-referencing canonical
- Distinguish missing from misdirected

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 484: write a sitemap audit check. Project 486 changes the surface problem while preserving the underlying method.

progress check

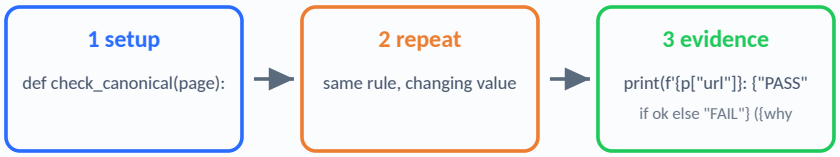
Reading target: write a canonical audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `def check_canonical(page):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'{p["url"]}: {"PASS" if ok else "FAIL"} ({why})')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def check_canonical(page):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if not c:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{p["url"]}: {"PASS" if ok else "FAIL"} ({why})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not c:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write a Canonical Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: check_canonical, page, get, canonical, url. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def check_canonical(page): | Core: if not c: | Evidence: print(f'{p["url"]}: {"PASS" if ok else "FAIL"} ({why})')

🛠️ Reader to Writer Bridge

1. Read: explain write a canonical audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 484.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 486 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{p["url"]}: {"PASS" if ok else "FAIL"} ({why})')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `def check_canonical(page):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write a canonical audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

The check returns pass plus a reason only when the canonical equals the page's own URL, distinguishing a self-reference from a missing or cross-pointing tag.

```
def check_canonical(page):
    c = page.get('canonical')
    url = page.get('url')
    if not c:
        return False, 'missing'
    if c != url:
        return False, 'points elsewhere'
    return True, 'self-referencing'
pages = [
    {'url': 'https://x.com/a', 'canonical': 'https://x.com/a'},
    {'url': 'https://x.com/b', 'canonical': 'https://x.com/a'},
    {'url': 'https://x.com/c', 'canonical': None},
]
for p in pages:
    ok, why = check_canonical(p)
    print(f'{p["url"]}: {"PASS" if ok else "FAIL"} ({why})')
```

```
output
https://x.com/a: PASS (self-referencing)
https://x.com/b: FAIL (points elsewhere)
https://x.com/c: FAIL (missing)
```

How it works:

1. check_canonical reads the page's canonical and url values from the page dictionary.
2. When no canonical exists it returns False with the reason string missing.
3. When the canonical differs from the URL it returns False with points elsewhere.
4. When they match it returns True with self-referencing as the reason.
5. Three sample pages are checked and each prints PASS or FAIL with its reason.

Try it:

1. Treat a canonical that differs only by trailing slash or protocol as a near-match warning.
2. Allow an intentional cross-page canonical when the page is flagged as a known duplicate.
3. Resolve relative canonical values against the page URL before comparing.
4. Normalise trailing slashes before comparing and re-test page b.

⚠ Common Mistakes

- A `None` canonical must fail as missing rather than crash on the equality test.
- Comparing canonicals ignoring trailing slashes would wrongly pass `/b/` vs `/b`.
- A page canonicalising to an equivalent but differently-cased URL still fails strict equality.

Skills gained canonical audit | tuple returns | self-reference check | indexability

→ Where You'll Use This: Auditing canonical tags catches duplicate-content and consolidation bugs that quietly bleed ranking signals or deindex pages, a frequent finding in large-site crawls.

⚡ Mental Model

A canonical tag is a page pointing at its own ID badge; point it at someone else's badge and the system files your work under their name.

Project 486 Write a Status-Code Audit Check

💡 The Idea Behind It

The status-code check confirms that key pages actually return 200, because redirects and errors all mean the indexable page is not being served as expected. check_status maps a code to a pass flag and a human label using range tests: exactly 200 passes, the three-hundred range is a redirect, the four-hundred range a client error, and anything else a server error. Bucketing by numeric range rather than listing individual codes keeps the function compact while still naming the failure class precisely.

What You'll Build & Learn

Write a status-code audit check that passes only on 200 and labels redirects and error classes otherwise.

- Bucket status codes by range
- Return a flag with a class label
- Treat anything but 200 as a failure

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 485: write a canonical audit check. Project 487 changes the surface problem while preserving the underlying method.

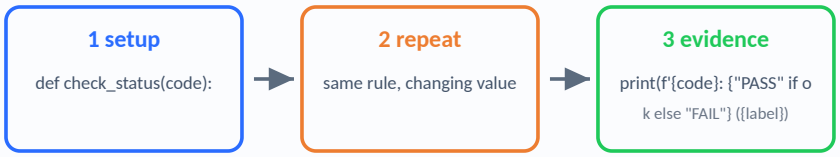
progress check

Reading target: write a status-code audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `def check_status(code):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'{code}: {"PASS" if ok else "FAIL"} ({label})')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def check_status(code):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if code == 200:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{code}: {"PASS" if ok else "FAIL"} ({label})')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if code == 200:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write a Status-Code Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: check_status, code, OK, redirect, client. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def check_status(code): | Core: if code == 200: | Evidence: print(f'{code}: {"PASS" if ok else "FAIL"} ({label})')

🛠️ Reader to Writer Bridge

1. Read: explain write a status-code audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 485.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 487 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{code}: {"PASS" if ok else "FAIL"} ({label})')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `def check_status(code):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write a status-code audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Bucketing the numeric code into 200, the 3xx, 4xx, or 5xx ranges returns a pass/fail plus a human label, the status leg of the audit.

```
def check_status(code):
    if code == 200:
        return True, 'OK'
    if 300 <= code < 400:
        return False, 'redirect'
    if 400 <= code < 500:
        return False, 'client error'
    return False, 'server error'
for code in [200, 301, 404, 503]:
    ok, label = check_status(code)
    print(f'{code}: {"PASS" if ok else "FAIL"} ({label})')
```

```
output
200: PASS (OK)
301: FAIL (redirect)
404: FAIL (client error)
503: FAIL (server error)
```

How it works:

1. check_status returns True with OK only when the code equals exactly 200.
2. A code in the three-hundred range returns False labelled redirect.
3. A code in the four-hundred range returns False labelled client error.
4. Any remaining code falls through to False labelled server error.
5. A loop checks 200, 301, 404 and 503, printing each code's verdict and label.

Try it:

1. Treat a 304 not-modified response as a pass for conditional requests.
2. Distinguish a 410 gone from other client errors since it signals intentional removal.
3. Return the full status reason phrase alongside the class label.
4. Add a 410 case and label it as gone.

⚠ Common Mistakes

- Testing `code == 300` instead of the `300 <= code < 400` range misses 301/302.
- Treating a 3xx as a soft pass hides that a key page should serve 200 directly.
- A 204 (no content) would fall through to server-error labelling here.

Skills gained status audit range bucketing labelled returns HTTP codes

→ Where You'll Use This: Verifying that priority URLs serve a clean 200 catches accidental redirects, soft errors and outages before they cost rankings, a core step in any crawl audit.

Mental Model

A status code is a receptionist's answer: only a clear yes, we have that, is a pass; sorry, moved, or it broke all turn the visitor away.

Project 487 Write a Core Web Vitals Audit Check

The Idea Behind It

The Core Web Vitals check confirms LCP, INP and CLS are all within Google's good thresholds, and one failing metric fails the whole check. check_cwv tests each metric against its limit, collects the names of the passing ones into a list, and returns a pass only when all three pass. Returning the list of passing metrics alongside the verdict is useful because it tells you not just that the check failed but precisely which vital, here CLS, dragged it down.

What You'll Build & Learn

Write a Core Web Vitals check that passes only when LCP, INP and CLS are all within good thresholds.

- Compare metrics against thresholds
- Collect passing names into a list
- Require all three to pass

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 486: write a status-code audit check. Project 488 changes the surface problem while preserving the underlying method.

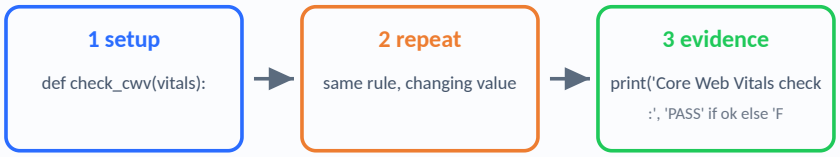
progress check

Reading target: write a core web vitals audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def check_cwv(vitals):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('Core Web Vitals check:', 'PASS' if ok else 'FAIL')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def check_cwv(vitals):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`passed = [n for n, ok in [('LCP', lcp_ok), ('INP', inp_ok), ('CLS', cls_ok)] if ok]`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('Core Web Vitals check:', 'PASS' if ok else 'FAIL')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `passed = [n for n, ok in [('LCP', lcp_ok), ('INP', inp_ok), ('CLS', cls_ok)] if ok]`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write a Core Web Vitals Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: check_cwv, vitals, lcp_ok, lcp_s, inp_ok. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def check_cwv(vitals): | Core: passed = [n for n, ok in [('LCP', lcp_ok), ('INP', inp_ok), ('CLS', cls... | Evidence: print('Core Web Vitals check:', 'PASS' if ok else 'FAIL')

🛠️ Reader to Writer Bridge

1. Read: explain write a core web vitals audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 486.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 488 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('Core Web Vitals check:', 'PASS' if ok else 'FAIL')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `def check_cwv(vitals):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write a core web vitals audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Comparing each metric to its threshold (LCP <= 2.5s, INP <= 200ms, CLS <= 0.1) and requiring all three to pass implements the all-or-nothing CWV verdict.

```
def check_cwv(vitals):
    lcp_ok = vitals['lcp_s'] <= 2.5
    inp_ok = vitals['inp_ms'] <= 200
    cls_ok = vitals['cls'] <= 0.1
    passed = [n for n, ok in [('LCP', lcp_ok), ('INP', inp_ok), ('CLS',
cls_ok)] if ok]
    return len(passed) == 3, passed
v = {'lcp_s': 2.1, 'inp_ms': 180, 'cls': 0.18}
ok, passed = check_cwv(v)
print('passing vitals:', passed)
print('Core Web Vitals check:', 'PASS' if ok else 'FAIL')
```

```
output
passing vitals: ['LCP', 'INP']
Core Web Vitals check: FAIL
```

How it works:

1. check_cwv reads a vitals dictionary and tests LCP, INP and CLS against their good limits.
2. Each metric's boolean is paired with its name in a list comprehension that keeps only passers.
3. The function returns True only when the passing list contains all three metrics.
4. A sample with a failing CLS is checked, yielding LCP and INP as the passing metrics.
5. Prints report the passing metrics list and the overall PASS or FAIL verdict.

Try it:

1. Return the failing metrics with their measured values and target thresholds.
2. Add a needs-improvement band so borderline metrics are not lumped with outright failures.
3. Weight the three vitals so a small CLS miss is not treated like a large LCP miss.
4. Lower CLS to 0.09 and watch the whole check pass.

⚠ Common Mistakes

- A CLS of 0.18 fails even though LCP and INP pass; the check is all-or-nothing.
- Using a `<` instead of `<=` would fail a metric sitting exactly on the threshold.

- Mixing up the units (seconds vs milliseconds) silently breaks the comparison.

Skills gained Core Web Vitals | threshold checks | list comprehension | all-pass logic

→ Where You'll Use This: A Core Web Vitals gate is a standard part of performance audits and CI checks, flagging the specific vital that needs work before a page can earn its good experience signal.

⚡ Mental Model

Core Web Vitals are a three-part medical panel: every reading must sit in the healthy range, because one bad result means the checkup is not passed.

Project 488 Write an Hreflang Audit Check

💡 The Idea Behind It

The hreflang check confirms a multilingual cluster has reciprocal annotations and an x-default, since missing return tags or no default break international targeting. check_hreflang short-circuits to a pass for monolingual pages, then for multilingual ones collects any language whose reciprocal flag is False and confirms an x-default is listed, passing only when nothing is missing and the default exists. Treating the monolingual case as not applicable rather than a failure is the nuance that keeps the audit fair on single-language sites.

What You'll Build & Learn

Write an hreflang audit check that passes when alternates reciprocate and an x-default is present.

- Short-circuit a not-applicable case
- Find non-reciprocating alternates
- Require an x-default annotation

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 487: write a core web vitals audit check. Project 489 changes the surface problem while preserving the underlying method.

progress check

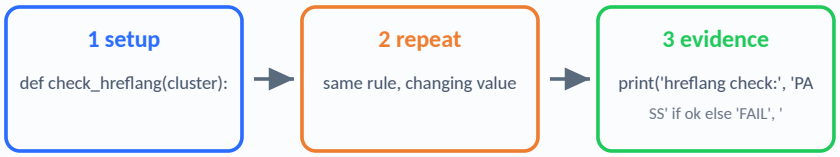
Reading target: write an hreflang audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `def check_hreflang(cluster):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.

- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('hreflang check:', 'PASS' if ok else 'FAIL', '|', why)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def check_hreflang(cluster):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if not langs:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('hreflang check:', 'PASS' if ok else 'FAIL', '|', why)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if not langs:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write an Hreflang Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: check_hreflang, cluster, langs, get, not. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: def check_hreflang(cluster): | Core: if not langs: | Evidence: print('hreflang check:', 'PASS' if ok else 'FAIL', '|', why)

🛠️ Reader to Writer Bridge

1. Read: explain write an hreflang audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 487.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 489 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('hreflang check:', 'PASS' if ok else 'FAIL', '|', why)`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `def check_hreflang(cluster):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write an hreflang audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

For a multilingual cluster the check requires every alternate to reciprocate and an x-default to be declared; a monolingual page is simply not applicable.

```
def check_hreflang(cluster):
    langs = cluster.get('langs', [])
    if not langs:
        return True, 'monolingual, n/a'
    # every alternate must reciprocate (point back)
    missing = [l for l, back in cluster['reciprocal'].items() if not back]
    has_xdefault = 'x-default' in langs
    ok = not missing and has_xdefault
    return ok, f'missing-return: {missing} xdefault: {has_xdefault}'
c = {'langs': ['en', 'de', 'x-default'], 'reciprocal': {'en': True, 'de':
True}}
ok, why = check_hreflang(c)
print('hreflang check:', 'PASS' if ok else 'FAIL', '|', why)
```

```
output
hreflang check: PASS | missing-return: [] xdefault: True
```

How it works:

1. check_hreflang reads the cluster's language list, returning a pass labelled monolingual when it is empty.
2. A comprehension collects every language whose reciprocal flag is False into a missing list.
3. It checks whether x-default appears among the declared languages.
4. The cluster passes only when nothing is missing and an x-default is present.
5. A sample cluster is checked and prints PASS with the empty missing list and x-default True.

Try it:

1. Validate that every declared language uses a well-formed locale code.
2. Detect a language that points to itself, which is an invalid self-reference.
3. Warn when an x-default points to a locale-specific page rather than a neutral one.
4. Set one locale's `reciprocal` to False and watch the check fail.

⚠ Common Mistakes

- A cluster with no `langs` is correctly treated as not applicable, not a failure.

- Checking only that tags exist, not that they reciprocate, misses one-way hreflang errors.
- Forgetting the x-default requirement passes clusters that lack a fallback locale.

Skills gained hreflang audit | reciprocity check | short-circuit logic | internationalisation

→ Where You'll Use This: Hreflang reciprocity and x-default bugs are common on multilingual sites and silently misroute users to the wrong locale, so this check is essential in international SEO audits.

⚡ Mental Model

Hreflang is a set of road signs between translated pages; if a sign only points one way, or there is no default exit, travellers get stranded.

Project 489 Write a Structured-Data Audit Check

💡 The Idea Behind It

The structured-data check confirms JSON-LD blocks parse and declare a @type, because invalid JSON or a missing type means engines cannot use the markup. check_structured_data attempts to json.loads each block inside a try, returning a failure on a ValueError, then rejects any object lacking an @type, and otherwise collects the types. Wrapping the parse in exception handling is the safeguard that lets the audit report invalid JSON-LD as a clean failure instead of crashing on the first malformed block.

What You'll Build & Learn

Write a structured-data check that passes only when every JSON-LD block parses and declares a @type.

- Guard json.loads with try-except
- Reject objects missing a key
- Collect parsed types for reporting

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 488: write an hreflang audit check. Project 490 changes the surface problem while preserving the underlying method.

progress check

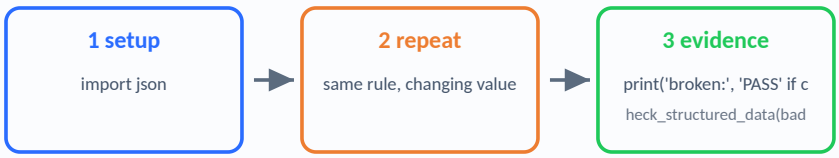
Reading target: write a structured-data audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import json` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('broken:', 'PASS' if check_structured_data(bad)[0] else 'FAIL', '|', check_structured_data(bad)[1])` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`import json`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if '@type' not in obj:`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('broken:', 'PASS' if check_structured_data(bad)[0] else 'FAIL', '|', check_structured_data(bad)[1])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if '@type' not in obj:`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Read the try path as the hoped-for route and the catch/except/finally path as the boundary plan.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write a Structured-Data Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: json, check_structured_data, blocks, parsed, obj. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route; exception handling defines failure paths.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: import json | Core: if '@type' not in obj: | Evidence: print('broken:', 'PASS' if check_structured_data(bad)[0] else 'FAIL', '...

🛠️ Reader to Writer Bridge

1. Read: explain write a structured-data audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 488.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 490 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('broken:', 'PASS' if check_structured_data(bad)[0] else 'FAIL', '|',`

`check_structured_data(bad)[1]`). Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import json` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write a structured-data audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Parsing each block with `json.loads` (catching `ValueError`) and requiring an `@type` validates the markup; the collected types report what the page exposes.

```
import json
def check_structured_data(blocks):
    parsed = []
    for b in blocks:
        try:
            obj = json.loads(b)
        except ValueError:
            return False, 'invalid JSON-LD'
        if '@type' not in obj:
            return False, 'missing @type'
        parsed.append(obj['@type'])
    return bool(parsed), f'types: {sorted(parsed)}'
good = ['{"@type": "Article"}', '{"@type": "FAQPage"}']
ok, why = check_structured_data(good)
print('structured data:', 'PASS' if ok else 'FAIL', '|', why)
bad = ['{"@type": "Article"', '{"name": "x"}']
print('broken:', 'PASS' if check_structured_data(bad)[0] else 'FAIL', '|',
check_structured_data(bad)[1])
```

```
output
structured data: PASS | types: ['Article', 'FAQPage']
broken: FAIL | invalid JSON-LD
```

How it works:

1. check_structured_data iterates the JSON-LD blocks, attempting json.loads on each one.
2. A ValueError during parsing returns a failure labelled invalid JSON-LD.
3. Any parsed object without an @type key returns a failure labelled missing @type.
4. Valid objects contribute their type to a list, and the function returns a pass when types were found.
5. A good pair of blocks prints PASS while a broken pair prints FAIL with invalid JSON-LD.

Try it:

1. Validate required properties for each type, such as a name on an Organization.
2. Handle a graph array of multiple objects within one script block.

3. Report every block's type and pinpoint exactly which block failed to parse.
4. Add a third valid block and watch the types list grow.

⚠ Common Mistakes

- Catching only `KeyError` and not `ValueError` would crash on truly invalid JSON.
- A block that parses but omits `@type` must fail even though the JSON is valid.
- Sorting the collected types keeps the printed order deterministic.

Skills gained structured-data audit | exception handling | JSON parsing | key validation

→ Where You'll Use This: Validating that JSON-LD actually parses and carries a type is a recurring rich-results audit step, catching the truncated or mistyped markup that quietly disqualifies a page.

Mental Model

JSON-LD is a customs form: if it is filled out wrong or missing the declaration line, the package, your rich result, gets refused at the gate.

Project 490 Write an On-Page Audit Check

The Idea Behind It

The on-page check confirms the table-stakes basics: a sane title length, a meta description in range, exactly one H1, and a viewport tag. check_onpage builds a dictionary of four named boolean results from length and count tests, sums the passers, and returns a pass only when all four hold along with the per-check detail. Returning the individual results dictionary, not just the verdict, is what lets the report show that the meta description failed while everything else passed.

What You'll Build & Learn

Write an on-page audit check covering title length, meta length, a single H1, and a viewport tag.

- Bound title and meta lengths
- Require exactly one H1
- Return per-check results plus verdict

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 489: write a structured-data audit check. Project 491 changes the surface problem while preserving the underlying method.

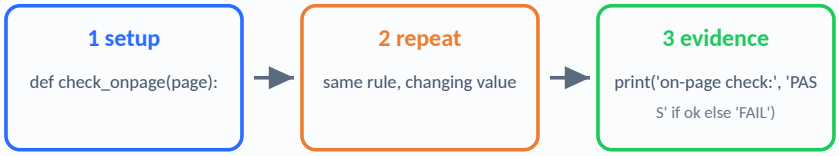
progress check

Reading target: write an on-page audit check is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `def check_onpage(page):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('on-page check:', 'PASS' if ok else 'FAIL')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def check_onpage(page):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`print(f' {name}: {"ok" if checks[name] else "fail"}')`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('on-page check:', 'PASS' if ok else 'FAIL')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `print(f' {name}: {"ok" if checks[name] else "fail"}')`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Write an On-Page Audit Check: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: check_onpage, page, checks, title, len. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def check_onpage(page): | Core: print(f' {name}: {"ok" if checks[name] else "fail"}') | Evidence: print('on-page check:', 'PASS' if ok else 'FAIL')

🛠️ Reader to Writer Bridge

1. Read: explain write an on-page audit check in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 489.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 491 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('on-page check:', 'PASS' if ok else 'FAIL')`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `def check_onpage(page):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of write an on-page audit check and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Each sub-check is a boolean over the page dict; the function passes only when all four hold, and the per-check printout shows exactly which basic failed.

```
def check_onpage(page):
    checks = {
        'title': 10 <= len(page.get('title', '')) <= 60,
        'meta_desc': 50 <= len(page.get('meta_desc', '')) <= 160,
        'single_h1': page.get('h1_count', 0) == 1,
        'has_viewport': page.get('viewport', False),
    }
    passed = sum(checks.values())
    return passed == len(checks), checks
page = {'title': 'Technical SEO Audit Guide', 'meta_desc': 'A short desc.',
'h1_count': 1, 'viewport': True}
ok, checks = check_onpage(page)
for name in sorted(checks):
    print(f'  {name}: {"ok" if checks[name] else "fail"}')
print('on-page check:', 'PASS' if ok else 'FAIL')
```

```
output
  has_viewport: ok
  meta_desc: fail
  single_h1: ok
  title: ok
on-page check: FAIL
```

How it works:

1. check_onpage builds a checks dictionary with four named boolean results for a page.
2. Title and meta tests bound their character lengths, and the H1 test requires a count of exactly one.
3. passed sums the truthy values, and the function passes only when all four checks hold.
4. A sample page with a short meta description is checked, failing that one item.
5. Sorted check names print ok or fail, then the overall on-page verdict prints.

Try it:

1. Add a check that the title is unique across the crawled page set.
2. Warn when the H1 text duplicates the title verbatim rather than complementing it.
3. Confirm the meta description contains the page's primary keyword.

4. Lengthen the meta description into the 50-160 band and watch the check pass.

⚠ Common Mistakes

- A 12-character meta description fails the 50-160 band even though a title that short would pass its own 10-60 band.
- Counting two H1s as a pass ignores that a page should have exactly one.
- A missing viewport key would be falsy and fail rather than crash thanks to `.get` defaults.

Skills gained on-page audit | length bounding | H1 counting | result reporting

→ Where You'll Use This: On-page basics are the most common findings in real audits, and a reusable check like this lets you scan thousands of pages for missing or malformed titles, metas and H1s.

Mental Model

On-page basics are a uniform inspection: title, description, one headline and a viewport are the buttons that must all be done up before the page steps out.

Project 491 Group the Crawlability Checks

The Idea Behind It

The full audit is assembled from groups, and this first group bundles eight crawlability points covering robots health, sitemap validity, orphans and internal linking. crawlability_points builds a list of named checks from a site dictionary, including a not-disallow-all test and bounded sitemap and crawl-delay tests, then returns both the count of passers and the labelled list. Returning the detailed checks alongside the score is the pattern that lets the printed report show seven of eight passing with the orphan check as the lone failure.

What You'll Build & Learn

Group eight crawlability checks, score how many pass, and print a checkbox report for a site fixture.

- Bundle related checks into a group
- Negate a flag inside a check
- Return both a score and details

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 490: write an on-page audit check. Project 492 changes the surface problem while preserving the underlying method.

progress check

Reading target: group the crawlability checks is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `def crawlability_points(s):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'crawlability: {pts}/8')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def crawlability_points(s):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`return sum(1 for _, ok in checks if ok), checks`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'crawlability: {pts}/8')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return sum(1 for _, ok in checks if ok), checks`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Group the Crawlability Checks: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: crawlability_points, checks, robots, reachable, robots_reachable. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def crawlability_points(s): | Core: return sum(1 for _, ok in checks if ok), checks | Evidence: print(f'crawlability: {pts}/8')

🛠️ Reader to Writer Bridge

1. Read: explain group the crawlability checks in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 490.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 492 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'crawlability: {pts}/8')`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `def crawlability_points(s):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of group the crawlability checks and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Collecting eight crawlability booleans into a list of (name, ok) pairs lets the group both score itself out of 8 and report exactly which point failed.

```
def crawlability_points(s):
    checks = [
        ('robots reachable', s['robots_reachable']),
        ('not disallow-all', not s['disallow_all']),
        ('crawl-delay sane', s['crawl_delay'] <= 10),
        ('sitemap present', s['sitemap_present']),
        ('sitemap within 50k', s['sitemap_urls'] <= 50000),
        ('sitemap valid xml', s['sitemap_valid']),
        ('no orphan key pages', s['orphans'] == 0),
        ('internal links ok', s['avg_internal_links'] >= 1),
    ]
    return sum(1 for _, ok in checks if ok), checks
site = {'robots_reachable': True, 'disallow_all': False, 'crawl_delay': 5,
        'sitemap_present': True, 'sitemap_urls': 3000, 'sitemap_valid': True,
        'orphans': 2, 'avg_internal_links': 4}
pts, checks = crawlability_points(site)
for name, ok in checks:
    print(f'[{"x" if ok else " "}] {name}')
print(f'crawlability: {pts}/8')
```

```
output
[x] robots reachable
[x] not disallow-all
[x] crawl-delay sane
[x] sitemap present
[x] sitemap within 50k
[x] sitemap valid xml
[ ] no orphan key pages
[x] internal links ok
crawlability: 7/8
```

How it works:

1. crawlability_points builds a list of eight named checks from a site dictionary.
2. Tests include robots reachability, a negated disallow-all flag, and bounded crawl-delay and sitemap size.
3. Orphan and internal-link tests compare counts against zero and a minimum.
4. The function returns the number of passing checks plus the full labelled list.

5. A site fixture is scored, printing a checkbox per check and seven out of eight overall.

Try it:

1. Add a check for an excessive crawl-delay that throttles indexing.
2. Weight orphan pages more heavily since they are invisible to crawlers entirely.
3. Return the failing check names so the group feeds a remediation list directly.
4. Set `orphans` to 0 and watch the group hit 8/8.

⚠ Common Mistakes

- Mixing a boolean signal with a numeric one (orphans count) needs the right comparison per check.
- Returning only the score loses the per-check detail the audit report needs.
- A `crawl_delay` over 10 seconds fails the politeness sub-check even with everything else green.

Skills gained check grouping | flag negation | score and detail | crawlability

→ Where You'll Use This: Grouping checks into crawlability, indexability, performance and content sections is how scalable audit tools stay organised, letting you report a sub-score per pillar.

Mental Model

This group is the access-and-discovery inspection: can crawlers get in, find the map, and reach every room without any locked-off orphan corridors.

Project 492 Group the Indexability and Status Checks

The Idea Behind It

The second audit group covers eight indexability and status points: canonicals, noindex, HTTPS, status codes, redirect length, hreflang and broken links. indexability_points builds the labelled checks from a site dictionary, negating the noindex flag and bounding redirect hops, then returns the passing count and the detailed list. Mirroring the structure of the crawlability group means the four pillar functions stay interchangeable, which is what later lets a capstone sum their scores into one audit total.

What You'll Build & Learn

Group eight indexability and status checks, score the passes, and print a report for a site fixture.

- Negate noindex within a check
- Bound redirect-hop length
- Keep group functions parallel

Skill Roadmap

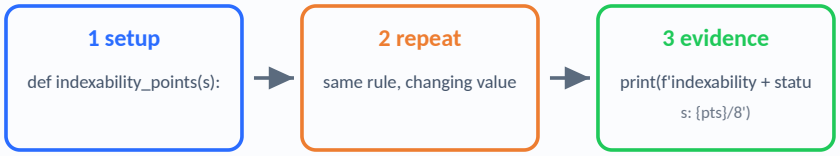
Skill step: pushes the same idea into a less automatic situation. It builds on Project 491: group the crawlability checks. Project 493 changes the surface problem while preserving the underlying method. *progress check*

Reading target: group the indexability and status checks is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def indexability_points(s):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'indexability + status: {pts}/8')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def indexability_points(s):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`return sum(1 for _, ok in checks if ok), checks`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'indexability + status: {pts}/8')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return sum(1 for _, ok in checks if ok), checks`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Group the Indexability and Status Checks: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: indexability_points, checks, canonical, self, ref. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def indexability_points(s): | Core: return sum(1 for _, ok in checks if ok), checks | Evidence: print(f'indexability + status: {pts}/8')

🛠️ Reader to Writer Bridge

1. Read: explain group the indexability and status checks in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 491.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 493 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'indexability + status: {pts}/8')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def indexability_points(s):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of group the indexability and status checks and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

The same (name, ok) pattern bundles indexability and status into a second group of 8 that scores itself and exposes the two failing points here.

```
def indexability_points(s):
    checks = [
        ('canonical self-ref', s['canonical_self']),
        ('no noindex on key pages', not s['noindex_key']),
        ('https everywhere', s['https']),
        ('all key pages 200', s['non_200'] == 0),
        ('redirect chains <= 3', s['max_redirect_hops'] <= 3),
        ('hreflang reciprocal', s['hreflang_reciprocal']),
        ('x-default present', s['x_default']),
        ('no broken links', s['broken_links'] == 0),
    ]
    return sum(1 for _, ok in checks if ok), checks
site = {'canonical_self': True, 'noindex_key': False, 'https': True, 'non_200':
0,
        'max_redirect_hops': 2, 'hreflang_reciprocal': True, 'x_default':
False, 'broken_links': 1}
pts, checks = indexability_points(site)
for name, ok in checks:
    print(f'[{"x" if ok else " "}] {name}')
print(f'indexability + status: {pts}/8')
```

```
output
[x] canonical self-ref
[x] no noindex on key pages
[x] https everywhere
[x] all key pages 200
[x] redirect chains <= 3
[x] hreflang reciprocal
[ ] x-default present
[ ] no broken links
indexability + status: 6/8
```

How it works:

1. indexability_points builds eight named checks from a site dictionary.
2. Tests cover self-canonical, a negated noindex flag, HTTPS, and a zero non-200 count.
3. Further checks bound redirect hops at three and confirm hreflang reciprocity and x-default.
4. A broken-links test passes only when the count is exactly zero.
5. A fixture is scored, printing each checkbox and six out of eight overall.

Try it:

1. Tighten the redirect-hop limit to two and warn on any chain over one.
2. Add a mixed-content check that fails HTTPS pages loading insecure assets.
3. Report the count of broken links so a single dead link reads differently from many.
4. Set `broken_links` to 0 and `x_default` to True for a perfect 8/8.

⚠ Common Mistakes

- `non_200 == 0` is the right test; counting `>= 0` would always pass.
- A redirect chain of exactly 3 passes the `<= 3` bound; off-by-one here changes the verdict.
- Treating one broken link as acceptable would let a real navigation bug slip through.

Skills gained indexability audit | flag negation | hop bounding | status checks

→ Where You'll Use This: Indexability is where rankings are won or lost, so a grouped check covering canonicals, noindex, redirects and broken links is core to any site-health monitoring setup.

Mental Model

If crawlability is getting in the door, this group is whether each room is open for filing: indexed, served cleanly, and not pointing visitors down dead ends.

Project 493 Group the Performance Checks

The Idea Behind It

The third audit group is eight performance points: the three Core Web Vitals plus mobile-friendliness, compression, render-blocking resources, page weight and TTFB. performance_points builds the labelled checks from a site dictionary, bounding each metric against its threshold, then returns the passing count and the detailed list. Folding the vitals in with broader performance signals like TTFB and page weight gives a fuller speed picture than the vitals alone, which is why CLS failing here still leaves seven of eight passing.

What You'll Build & Learn

Group eight performance checks spanning the vitals and broader speed signals, score them, and report.

- Combine vitals with broader metrics

- Bound each metric to a threshold
- Score a performance pillar

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 492: group the indexability and status checks. Project 494 changes the surface problem while preserving the underlying method.

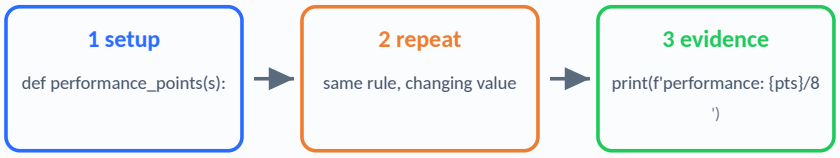
progress check

Reading target: group the performance checks is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `def performance_points(s):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'performance: {pts}/8')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def performance_points(s):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`return sum(1 for _, ok in checks if ok), checks`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(f'performance: {pts}/8')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return sum(1 for _, ok in checks if ok), checks`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Group the Performance Checks: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: performance_points, checks, lcp_s, ms, inp_ms. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def performance_points(s): | Core: return sum(1 for _, ok in checks if ok), checks | Evidence: print(f'performance: {pts}/8')

Reader to Writer Bridge

1. Read: explain group the performance checks in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 492.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 494 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'performance: {pts}/8')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def performance_points(s):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of group the performance checks and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Bundling the vitals with delivery metrics into a third group of 8 keeps performance in one scorable unit that names the one failing metric (CLS).

```
def performance_points(s):
    checks = [
        ('LCP <= 2.5s', s['lcp_s'] <= 2.5),
        ('INP <= 200ms', s['inp_ms'] <= 200),
        ('CLS <= 0.1', s['cls'] <= 0.1),
        ('mobile-friendly', s['mobile_friendly']),
        ('compression on', s['compressed']),
        ('render-blocking <= 2', s['render_blocking'] <= 2),
        ('page weight <= 2MB', s['weight_kb'] <= 2048),
        ('TTFB <= 800ms', s['ttfb_ms'] <= 800),
    ]
    return sum(1 for _, ok in checks if ok), checks
site = {'lcp_s': 2.3, 'inp_ms': 190, 'cls': 0.12, 'mobile_friendly': True,
        'compressed': True, 'render_blocking': 1, 'weight_kb': 1500, 'ttfb_ms':
600}
pts, checks = performance_points(site)
for name, ok in checks:
    print(f'[{"x" if ok else " "}] {name}')
print(f'performance: {pts}/8')
```

```
output
[x] LCP <= 2.5s
[x] INP <= 200ms
[ ] CLS <= 0.1
[x] mobile-friendly
[x] compression on
```

```
[x] render-blocking <= 2
[x] page weight <= 2MB
[x] TTFB <= 800ms
performance: 7/8
```

How it works:

1. performance_points builds eight named checks from a site dictionary.
2. The first three checks bound LCP, INP and CLS against their good thresholds.
3. Further checks cover mobile-friendliness, compression, render-blocking count, weight and TTFB.
4. The function returns the count of passing checks plus the labelled list.
5. A fixture with a high CLS is scored, printing each checkbox and seven out of eight.

Try it:

1. Add a check for an excessive number of third-party scripts.
2. Tighten the page-weight limit for pages targeting mobile-first audiences.
3. Report the worst-performing metric so optimisation effort goes where it counts most.
4. Drop CLS to 0.08 and watch the group reach 8/8.

⚠ Common Mistakes

- CLS just over 0.1 fails while the other seven pass; one metric sinks the point, not the group.
- Comparing page weight in the wrong unit (bytes vs KB) silently mis-scores it.
- A render-blocking count of exactly 2 passes the `<= 2` bound.

Skills gained performance audit | metric bounding | vitals plus speed | pillar scoring

→ Where You'll Use This: A grouped performance check that goes beyond the three vitals is what page-speed audits use to explain why a fast-feeling page still has a heavy payload or slow server response.

Mental Model

This group is the speed-and-stability dyno test: not just the headline lap times of the vitals but the engine weight, fuel delivery and how fast it leaves the line.

Project 494 Group the On-Page, AEO and E-E-A-T Checks

The Idea Behind It

The fourth audit group is eight content points spanning on-page, AEO and E-E-A-T: title and meta lengths, a single H1, valid schema, a concise lead answer, FAQ or HowTo schema, an author bio and freshness. content_points builds these labelled checks from a site dictionary, with the lead-answer test requiring a positive word count at or below sixty, and returns the score plus the detail. Bundling traditional on-page basics

together with answer-engine and trust signals reflects how a modern audit treats content quality as one combined pillar.

What You'll Build & Learn

Group eight content checks across on-page, AEO and E-E-A-T signals, score them, and print a report.

- Mix on-page, AEO and E-E-A-T
- Bound a lead-answer word count
- Score a combined content pillar

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 493: group the performance checks. Project 495 changes the surface problem while preserving the underlying method.

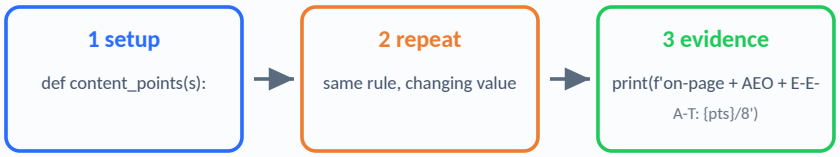
progress check

Reading target: group the on-page, aeo and e-e-a-t checks is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `def content_points(s):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'on-page + AEO + E-E-A-T: {pts}/8')` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def content_points(s):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`return sum(1 for _, ok in checks if ok), checks`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'on-page + AEO + E-E-A-T: {pts}/8')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return sum(1 for _, ok in checks if ok), checks`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Group the On-Page, AEO and E-E-A-T Checks: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: content_points, checks, title, chars, title_len. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def content_points(s): | Core: return sum(1 for _, ok in checks if ok), checks | Evidence: print(f'on-page + AEO + E-E-A-T: {pts}/8')

🛠️ Reader to Writer Bridge

1. Read: explain group the on-page, aeo and e-e-a-t checks in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 493.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 495 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'on-page + AEO + E-E-A-T: {pts}/8')`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def content_points(s):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of group the on-page, aeo and e-e-a-t checks and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

This group fuses classic on-page checks with the GEO/AEO and E-E-A-T signals from earlier in the chapter into the final scorable group of 8.

```
def content_points(s):
    checks = [
        ('title 10-60 chars', 10 <= s['title_len'] <= 60),
        ('meta desc 50-160', 50 <= s['meta_len'] <= 160),
        ('single H1', s['h1_count'] == 1),
        ('valid schema', s['valid_schema']),
        ('concise lead answer', s['lead_answer_words'] <= 60 and
s['lead_answer_words'] > 0),
        ('FAQ/HowTo schema', s['aeo_schema']),
        ('author bio', s['author_bio']),
        ('content fresh', s['fresh']),
    ]
    return sum(1 for _, ok in checks if ok), checks
site = {'title_len': 42, 'meta_len': 120, 'h1_count': 1, 'valid_schema': True,
        'lead_answer_words': 38, 'aeo_schema': True, 'author_bio': True,
'fresh': False}
pts, checks = content_points(site)
for name, ok in checks:
```

```
    print(f'[{"x" if ok else " "}] {name}')
print(f'on-page + AEO + E-E-A-T: {pts}/8')
```

```
output
[x] title 10-60 chars
[x] meta desc 50-160
[x] single H1
[x] valid schema
[x] concise lead answer
[x] FAQ/HowTo schema
[x] author bio
[ ] content fresh
on-page + AEO + E-E-A-T: 7/8
```

How it works:

1. content_points builds eight named checks from a site dictionary.
2. Title and meta length tests sit beside a single-H1 and a valid-schema check.
3. The lead-answer check requires a positive word count at or below sixty words.
4. Further checks cover AEO schema, an author bio, and content freshness.
5. A fixture failing only freshness is scored, printing each checkbox and seven out of eight.

Try it:

1. Split the pillar into separate on-page and AEO sub-scores for finer reporting.
2. Add an entity-coverage check drawn from the earlier entity-counting project.
3. Require the lead answer to also exceed a minimum so empty leads do not pass.
4. Flip `fresh` to True and watch the group hit 8/8.

⚠ Common Mistakes

- A lead answer of 0 words must fail even though `<= 60` alone would pass; both bounds are needed.
- Title length is character-based, so a 9-character title fails the 10-60 band.
- Freshness here is a precomputed boolean; deriving it from a date belongs in an earlier step.

Skills gained content audit | AEO signals | word-count bounding | pillar scoring

→ Where You'll Use This: As answer engines reshape SEO, audits increasingly fold AEO and E-E-A-T signals into the content pillar, and this grouped check is how that combined quality view gets scored.

Mental Model

This group grades the writing itself: is it well-labelled, quotable by an engine, backed by a credible author, and recently kept up to date.

Project 495 Map an Audit Score to a Letter Grade

The Idea Behind It

A thirty-two point score is far easier to act on as a letter grade, so this project maps percentage bands to A through F. The grade function divides the score by a default total of thirty-two, scales to a percentage, and returns a letter using descending threshold checks where the first satisfied band wins. Ordering the comparisons from highest to lowest is the trick that lets a simple chain of returns replace nested conditionals while still assigning each score to exactly one grade band.

What You'll Build & Learn

Map a thirty-two point audit score to an A-through-F letter grade using percentage bands.

- Convert a score to a percentage
- Order thresholds high to low
- Return on the first matching band

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 494: group the on-page, aeo and e-e-a-t checks. Project 496 changes the surface problem while preserving the underlying method.

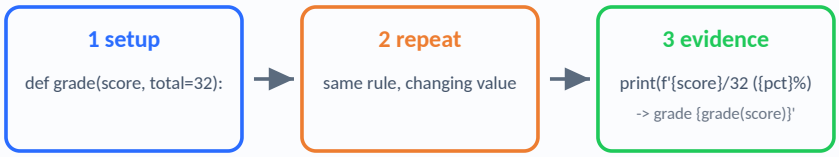
progress check

Reading target: map an audit score to a letter grade is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def grade(score, total=32):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'{score}/32 ({pct}%) -> grade {grade(score)}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def grade(score, total=32):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`if pct >= 90: return 'A'`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(f'{score}/32 ({pct}%) -> grade {grade(score)}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `if pct >= 90: return 'A'`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Comparison/logic: Decides truth, branches, filters, or loop continuation.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Map an Audit Score to a Letter Grade: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: grade, score, total, pct, round. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def grade(score, total=32): | Core: if pct >= 90: return 'A' | Evidence: print(f'{score}/32 ({pct}%) -> grade {grade(score)}')

🛠️ Reader to Writer Bridge

1. Read: explain map an audit score to a letter grade in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 494.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 496 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{score}/32 ({pct}%) -> grade {grade(score)}')`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `def grade(score, total=32):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of map an audit score to a letter grade and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Converting the score to a percentage and bucketing into 90/80/70/60 bands yields the familiar A-F grade used in the final audit report.

```
def grade(score, total=32):
    pct = 100 * score / total
    if pct >= 90: return 'A'
    if pct >= 80: return 'B'
    if pct >= 70: return 'C'
    if pct >= 60: return 'D'
```

```
    return 'F'
for score in [31, 27, 23, 20, 14]:
    pct = round(100 * score / 32)
    print(f'{score}/32 ({pct}%) -> grade {grade(score)}')
```

```
output
31/32 (97%) -> grade A
27/32 (84%) -> grade B
23/32 (72%) -> grade C
20/32 (62%) -> grade D
14/32 (44%) -> grade F
```

How it works:

1. The grade function takes a score and a default total of thirty-two.
2. It computes the percentage as one hundred times the score over the total.
3. Descending if-returns check ninety, eighty, seventy and sixty for grades A through D.
4. Any percentage below sixty falls through to grade F.
5. A loop grades five sample scores, printing each score, its rounded percentage, and its letter.

Try it:

1. Add plus and minus modifiers so a high B reads differently from a low one.
2. Make the band thresholds parameters so different audit rubrics can be applied.
3. Return both the letter and a short descriptive label such as healthy or critical.
4. Change the A cutoff to 95% and watch the 31/32 case fall to B.

⚠ Common Mistakes

- Using `>` instead of `>=` at the band edges drops a score sitting exactly on 90%.
- Computing the percentage with integer division would truncate and shift grades.
- Hard-coding `total=32` means reusing this for a different scale needs the argument passed.

Skills gained grade mapping | percentage conversion | threshold ordering | early returns

→ Where You'll Use This: Translating a raw audit score into a letter grade is how findings get communicated to non-technical stakeholders, turning a number into an at-a-glance site-health verdict.

Mental Model

This is a school report card: the long list of marks collapses into one letter a parent, or a client, instantly understands.

Project 496 Run the Full 32-Point Technical SEO Audit

The Idea Behind It

The capstone runs all thirty-two checks over one site fixture, scores it, assigns a letter grade, and lists every failing point. The code stores thirty-two boolean results in a single dictionary, partitions the keys into passed and failed comprehensions, counts the passers as the score, and inlines the grade banding. Driving both the score and the failure list from one dictionary of results is what makes this the chapter's complete audit, since every earlier check now feeds one consolidated verdict.

What You'll Build & Learn

Run the complete thirty-two point audit over one site fixture, score and grade it, and list every failure.

- Hold all results in one dict
- Partition keys into pass and fail
- Inline the grade banding

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 495: map an audit score to a letter grade. Project 497 changes the surface problem while preserving the underlying method.

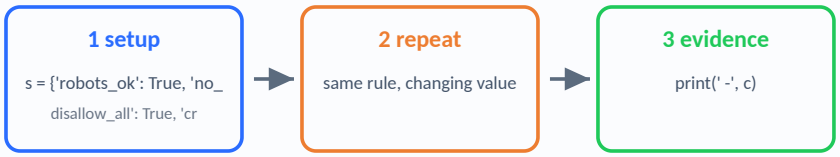
progress check

Reading target: run the full 32-point technical seo audit is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `s = {'robots_ok': True, 'no_disallow_all': True, 'crawl_delay_ok': True, 'sitemap_present': True,` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(' –', c)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. visual trace

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`s = {'robots_ok': True, 'no_disallow_all': True, 'crawl_delay_ok': True, 'sitemap_present': True,`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`passed = [c for c in checks if s[c]]`	This line controls repeated work. Name what changes each pass and what stays fixed. slow line
3. Check the evidence	`print(' –', c)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `passed = [c for c in checks if s[c]]`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Run the Full 32-Point Technical SEO Audit: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: robots_ok, no_disallow_all, crawl_delay_ok, sitemap_present, sitemap_size_ok. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Collection data: list/array/map/set shape determines lookup, order, and uniqueness.
Evidence	Entry: s = {'robots_ok': True, 'no_disallow_all': True, 'crawl_delay_ok': True... | Core: passed = [c for c in checks if s[c]] | Evidence: print(' -', c)

🛠️ Reader to Writer Bridge

1. Read: explain run the full 32-point technical seo audit in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 495.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 497 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(' –', c)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `s = {'robots_ok': True, 'no_disallow_all': True, 'crawl_delay_ok': True, 'sitemap_present': True,` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of run the full 32-point technical seo audit and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

   same idea, new surface

🔍 From Concept to Code (Python)

Flattening all 32 boolean checks into one dict lets a single pass count passes, derive the percentage and grade, and emit the exact list of failures to fix.

```
s = {'robots_ok': True, 'no_disallow_all': True, 'crawl_delay_ok': True,
'sitemap_present': True,
     'sitemap_size_ok': True, 'sitemap_valid': True, 'no_orphans': False,
'internal_links_ok': True,
     'canonical_self': True, 'no_noindex_key': True, 'https': True, 'all_200':
```

```
True,
     'redirects_short': True, 'hreflang_reciprocal': True, 'x_default': False,
'no_broken': True,
     'lcp_ok': True, 'inp_ok': True, 'cls_ok': False, 'mobile_ok': True,
     'compressed': True, 'render_block_ok': True, 'weight_ok': True, 'ttfb_ok':
True,
     'title_ok': True, 'meta_ok': True, 'h1_ok': True, 'schema_valid': True,
     'lead_answer_ok': True, 'aeo_schema': True, 'author_bio': True, 'fresh':
False}
checks = list(s.keys())
passed = [c for c in checks if s[c]]
failed = [c for c in checks if not s[c]]
score = len(passed)
pct = round(100 * score / 32)
g = 'A' if pct >= 90 else 'B' if pct >= 80 else 'C' if pct >= 70 else 'D' if
pct >= 60 else 'F'
print('=== 32-POINT TECHNICAL SEO AUDIT ===')
print(f'score: {score}/32 ({pct}%)  grade: {g}')
print(f'failures ({len(failed)}):')
for c in failed:
    print('  -', c)
```

```
output
=== 32-POINT TECHNICAL SEO AUDIT ===
score: 28/32 (88%)  grade: B
failures (4):
  - no_orphans
  - x_default
  - cls_ok
  - fresh
```

How it works:

1. A single dictionary records thirty-two boolean check results for one site fixture.
2. Comprehensions split the keys into a passed list and a failed list by their value.
3. The score is the length of the passed list, and the percentage is rounded against thirty-two.
4. An inline chained conditional maps the percentage to a letter grade A through F.
5. A banner prints the score, percentage and grade, then loops to list each failing check.

Try it:

1. Group the failures by pillar so the report shows where the worst cluster of problems sits.
2. Read the site results from a real crawl rather than a hard-coded fixture.
3. Emit the result as JSON so the audit can feed a dashboard or alert.
4. Flip `cls_ok` and `fresh` to True and watch the grade climb to A.

⚠ Common Mistakes

- The dict must hold exactly 32 keys or the `/32` denominator no longer matches reality.
- Iterating `s.items()` for failures preserves insertion order; sorting would reorder the report.
- A boolean stored as the string `'False'` would be truthy and silently pass.

Skills gained `full audit` `dict partitioning` `inline grading` `failure listing`

→ **Where You'll Use This:** A single function that scores a site against a fixed checklist and prints its failures is the deliverable clients want, and the foundation any automated audit dashboard is built on.

⚡ Mental Model

This is the full physical exam: every individual test from the chapter run at once, rolled into one chart with a grade and a list of what to treat.

Project 497 Turn Audit Failures Into a Remediation Plan

💡 The Idea Behind It

A list of failing check names is only useful if it tells you what to do, so this project maps each failure to a concrete fix. A FIXES dictionary pairs check names with remediation sentences, and the code loops over a list of failures, printing a numbered plan that looks up each fix and falls back to investigate manually for unknown names. Using dict.get with a default is the safeguard that keeps the plan from crashing on a failure the lookup table has not yet been taught to fix.

What You'll Build & Learn

Turn a list of audit failures into a numbered remediation plan by mapping each failure to a fix.

- Map failures to fix instructions
- Default unknown failures gracefully
- Number a plan with enumerate

🧭 Skill Roadmap

`Skill step:` pushes the same idea into a less automatic situation. It builds on Project 496: run the full 32-point technical seo audit. Project 498 changes the surface problem while preserving the underlying method.

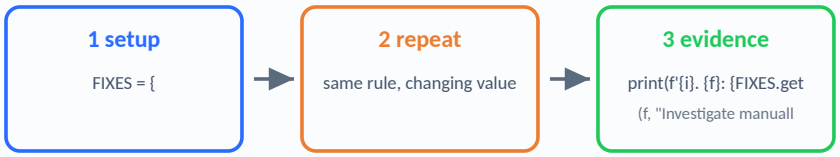
progress check

`Reading target:` turn audit failures into a remediation plan is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `FIXES = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The comprehension compresses a loop into a readable noun phrase: make this for each item if it qualifies. Expand it mentally when it feels dense.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print(f'{i}. {f}: {FIXES.get(f, "Investigate manually.")}')` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`FIXES = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`'x_default': 'Add an x-default hreflang for unmatched locales.',`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(f'{i}. {f}: {FIXES.get(f, "Investigate manually.")}')`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `'x_default': 'Add an x-default hreflang for unmatched locales.',`

Construction pattern

The construction repeats one rule over changing values. Identify the changing variable and the fixed rule before running it.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Expand the comprehension into an ordinary loop, then compress it back once you can explain each slot.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Turn Audit Failures Into a Remediation Plan: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: no_orphans, Add, internal, links, to. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: FIXES = { | Core: 'x_default': 'Add an x-default hreflang for unmatched locales.', | Evidence: print(f'{i}. {f}: {FIXES.get(f, "Investigate manually.")}')

🛠️ Reader to Writer Bridge

1. Read: explain turn audit failures into a remediation plan in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 496.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 498 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(f'{i}. {f}: {FIXES.get(f, "Investigate manually.")}')`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `FIXES = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of turn audit failures into a remediation plan and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

A fix dictionary keyed by check name converts each failure into a numbered, human-readable remediation step, with a fallback for unknown checks.

```
FIXES = {
    'no_orphans': 'Add internal links to orphan pages or include them in the
sitemap.',
    'x_default': 'Add an x-default hreflang for unmatched locales.',
    'cls_ok': 'Reserve space for images and embeds to stop layout shift.',
    'fresh': 'Refresh the content and update the lastmod date.',
}
failures = ['no_orphans', 'cls_ok', 'fresh']
print('REMEDIATION PLAN')
for i, f in enumerate(failures, 1):
    print(f'{i}. {f}: {FIXES.get(f, "Investigate manually.")}')
```

```
output
REMEDIATION PLAN
1. no_orphans: Add internal links to orphan pages or include them in the sitemap.
2. cls_ok: Reserve space for images and embeds to stop layout shift.
3. fresh: Refresh the content and update the lastmod date.
```

How it works:

1. A FIXES dictionary pairs each known check name with a plain-language remediation sentence.
2. A failures list names the three checks that did not pass this audit.
3. enumerate starting at one numbers the plan as it iterates the failures.
4. Each line looks up the fix with get, defaulting to investigate manually when unknown.
5. The printed output is a clean, numbered remediation plan.

Try it:

1. Attach an effort or impact estimate to each fix and sort the plan accordingly.
2. Expand the FIXES table to cover all thirty-two checks from the audit.
3. Group related fixes so a single engineering ticket can address several at once.
4. Add a fix entry for `x_default` and re-run with it in the list.

⚠ Common Mistakes

- A failure with no entry in `FIXES` falls back to a generic message rather than crashing.
- Enumerating from 0 instead of 1 produces an off-putting zero-indexed plan.
- Reordering the failures list changes the step numbering, so sort it first if stability matters.

Skills gained remediation mapping | dict get default | enumerate numbering | actionable audits

→ Where You'll Use This: Turning failures into prioritised fix instructions is what makes an audit actionable, and this lookup pattern is how audit tools generate the recommendation section of a report.

Mental Model

The audit is a diagnosis and this is the prescription: each problem named gets a specific instruction, so the patient knows exactly what to take and when.

Project 498 Diff Two Audits for Regressions

The Idea Behind It

Audits are most valuable over time, so this project diffs a baseline run against the latest to surface regressions and wins. A score helper sums passing booleans, and the code computes both totals, then derives regressions as checks that were True before and False now, and fixes as the reverse. Sorting the regression and fix lists makes the diff stable and readable, while the score delta gives a one-number summary of whether the site got healthier or worse since the baseline.

What You'll Build & Learn

Diff a baseline audit against the latest run to report the score delta, regressions, and newly fixed checks.

- Sum booleans for an audit score
- Derive regressions from two runs
- Compute a score delta

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 497: turn audit failures into a remediation plan. Project 499 changes the surface problem while preserving the underlying method.

progress check

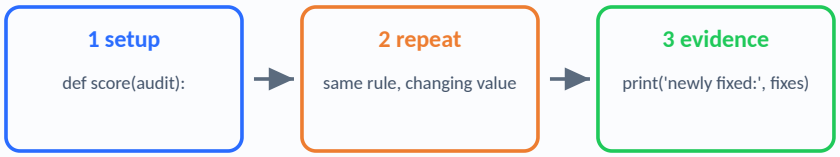
Reading target: diff two audits for regressions is the surface task; the transferable Python move is to mark setup, rule, and output before running.

setup -> rule -> result

Logic Lens

- Setup: `def score(audit):` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to turn a working idea into a reusable unit. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('newly fixed:', fixes)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`def score(audit):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`return sum(1 for v in audit.values() if v)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('newly fixed:', fixes)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `return sum(1 for v in audit.values() if v)`

Construction pattern

The construction is a callable unit: name inputs, perform a rule, return or print evidence. The boundary is what makes the idea reusable.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Write the contract in words: given these inputs, this block returns or prints this kind of result.

Rewrite drill

Call the same function with a different input and predict the returned or printed value.

Code Reading Stack

Layer	What to read
Intent	Diff Two Audits for Regressions: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: score, audit, sum, values, baseline. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: def score(audit): | Core: return sum(1 for v in audit.values() if v) | Evidence: print('newly fixed:', fixes)

🛠️ Reader to Writer Bridge

1. Read: explain diff two audits for regressions in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 497.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 499 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('newly fixed:', fixes)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def score(audit):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of diff two audits for regressions and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🔍 From Concept to Code (Python)

Scoring both audits and taking set-style differences over the shared keys identifies exactly which checks regressed and which were newly fixed since the baseline.

```
def score(audit):
    return sum(1 for v in audit.values() if v)
baseline = {'lcp_ok': True, 'cls_ok': True, 'schema_valid': True, 'fresh':
True, 'all_200': True}
latest =   {'lcp_ok': True, 'cls_ok': False, 'schema_valid': True, 'fresh':
True, 'all_200': False}
b, l = score(baseline), score(latest)
regressions = sorted(k for k in baseline if baseline[k] and not latest[k])
fixes = sorted(k for k in baseline if not baseline[k] and latest[k])
print(f'baseline {b}/{len(baseline)} -> latest {l}/{len(latest)} (delta {l -
b})')
print('regressions:', regressions)
print('newly fixed:', fixes)
```

```
output
baseline 5/5 -> latest 3/5 (delta -2)
regressions: ['all_200', 'cls_ok']
newly fixed: []
```

How it works:

1. A score helper sums the truthy values in any audit dictionary.
2. Baseline and latest dictionaries record the same five checks at two points in time.
3. Their scores are computed and the delta printed as latest minus baseline.
4. A comprehension finds checks that were True in baseline but False in latest as regressions.
5. A mirror comprehension finds newly fixed checks, and both sorted lists print.

Try it:

1. Handle checks present in one run but not the other rather than assuming identical keys.
2. Print the regressions with the fixes from the remediation table beside them.
3. Track the delta across many runs to plot a site-health trend line.
4. Fix `cls_ok` in `latest` and watch it move to the newly-fixed list.

⚠ Common Mistakes

- Comparing scores alone hides which checks changed; the per-key diff is what matters.

- Iterating only the latest dict misses checks present in the baseline but dropped later.
- Sorting the regression and fix lists keeps the diff output deterministic.

Skills gained audit diffing boolean summing regression detection trend tracking

→ **Where You'll Use This:** Diffing audits between deploys or months is how teams catch SEO regressions introduced by a release, making this comparison a natural fit for continuous monitoring.

Mental Model

Like comparing two blood-test results: the single number tells you better or worse, but the regression list shows exactly which marker slipped since last time.

Project 499 Aggregate an Audit Across Many Pages

The Idea Behind It

A site-wide audit rolls per-page results into pass rates per check and names the weakest page, showing where a problem is systemic versus isolated. The code iterates a fixed list of checks, counts how many pages pass each, and prints the rate as a fraction and percentage, then uses min with a key that sums each page's booleans to find the weakest page. Summing each page's passing checks as the sort key is what cleanly identifies the single page most in need of attention.

What You'll Build & Learn

Aggregate per-page audit results into per-check pass rates and identify the weakest page on the site.

- Compute per-check pass rates
- Iterate a fixed list of checks
- Find the weakest page with min

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 498: diff two audits for regressions. Project 500 changes the surface problem while preserving the underlying method.

progress check

Reading target: aggregate an audit across many pages is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `pages = {` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.

- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('weakest page:', worst)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`pages = {`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`hits = sum(1 for p in pages.values() if p[c])`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('weakest page:', worst)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `hits = sum(1 for p in pages.values() if p[c])`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Aggregate an Audit Across Many Pages: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: pages, home, title_ok, schema, lead_answer. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: pages = { | Core: hits = sum(1 for p in pages.values() if p[c]) | Evidence: print('weakest page:', worst)

🛠️ Reader to Writer Bridge

1. Read: explain aggregate an audit across many pages in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 498.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, Project 500 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('weakest page:', worst)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `pages = {` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of aggregate an audit across many pages and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Counting how many pages pass each check gives per-check pass rates, and summing each page's passes (with a sorted tiebreak) finds the weakest page to prioritise.

```
pages = {
    '/home': {'title_ok': True, 'schema': True, 'lead_answer': True, 'fresh':
True},
    '/blog/a': {'title_ok': True, 'schema': False, 'lead_answer': True,
'fresh': False},
    '/blog/b': {'title_ok': False, 'schema': False, 'lead_answer': True,
'fresh': True},
}
checks = ['title_ok', 'schema', 'lead_answer', 'fresh']
n = len(pages)
print('site-wide pass rate per check:')
for c in checks:
    hits = sum(1 for p in pages.values() if p[c])
    print(f'  {c}: {hits}/{n} ({round(100 * hits / n)}%)')
worst = min(sorted(pages), key=lambda p: sum(pages[p].values()))
print('weakest page:', worst)
```

```
output
site-wide pass rate per check:
  title_ok: 2/3 (67%)
  schema: 1/3 (33%)
  lead_answer: 3/3 (100%)
  fresh: 2/3 (67%)
weakest page: /blog/a
```

How it works:

1. A pages dictionary records four boolean checks for each of three pages.
2. For each check in a fixed list, the code counts how many pages pass it.
3. Each pass rate prints as a fraction of pages and a rounded percentage.
4. min with a key that sums each page's check values finds the weakest page.
5. The weakest page path is printed after the per-check pass rates.

Try it:

1. Report the worst-performing check across the whole site, not only the worst page.
2. Weight pages by traffic so a popular weak page outranks an obscure one.
3. Output the per-page and per-check matrix as a CSV for further analysis.

4. Add a page that fails every check and watch it become the weakest.

⚠ Common Mistakes

- A tie for weakest page resolves by sorted URL order, so `min` over `sorted(pages)` is deterministic.
- Computing pass rate over a subset of pages, not all, would misstate the percentage.
- Summing boolean values relies on True==1; a stray string value would break the count.

Skills gained site-wide audit | pass-rate aggregation | min with key | weakest page

→ Where You'll Use This: Rolling per-page checks into site-wide pass rates is how large-site audits separate a systemic schema gap from a one-off bad page, focusing remediation where it scales.

⚡ Mental Model

This is the class gradebook view: pass rates per question show which topic the whole class struggled with, while the lowest total flags the student needing help.

Project 500 Build a Combined GEO and Technical Scorecard

💡 The Idea Behind It

The final scorecard fuses the thirty-two point technical audit with a five-point GEO and AEO readiness score into one holistic number. The code takes the technical score as given, counts the passing GEO signals from a dictionary, adds the two scores and their two totals, computes a combined percentage, assigns a letter grade, and surfaces the single top gap. Combining classic technical health with answer-engine readiness in one figure is the chapter's culminating idea, reflecting how a modern audit must judge both at once.

What You'll Build & Learn

Fuse the technical audit score with a GEO and AEO readiness score into one combined grade and top gap.

- Combine two score dimensions
- Sum scores and totals together
- Surface the single top gap

🧭 Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 499: aggregate an audit across many pages. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: build a combined geo and technical scorecard is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `technical = 28 # out of 32` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The loop supplies repeated chances and the condition decides which chances count. That is the classic filter/count/accumulate shape. *repeat, test, update*
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
- Evidence: `print('top gap:', sorted(k for k, v in geo.items() if not v)[0])` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the chapter idea, and evidence proves your trace against the output. *visual trace*

📝 Worked Trace

Move	Code to inspect	Why it matters
1. Set up	`technical = 28 # out of 32`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
2. Apply the rule	`geo_score = sum(1 for v in geo.values() if v)`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print('top gap:', sorted(k for k, v in geo.items() if not v)[0])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: GEO, AEO and the End-to-End Technical Audit becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `geo_score = sum(1 for v in geo.values() if v)`

Construction pattern

The construction is repeat -> test -> update. Trace one pass first, then ask which passes are included or skipped.

Syntax tokens to inspect

Parentheses: Group function arguments, method parameters, conditions, or arithmetic subexpressions. Braces: Mark object literals, blocks, class bodies, CSS declarations, or interpolation boundaries depending on the language. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or assigns CSS properties to values. Dot: Reads as ownership or access: object.method, module.function, package.name, or selector.class.

Build-it-yourself move

Create a two-column trace: loop value on the left, changed result or side effect on the right.

Rewrite drill

Change the loop range or collection by one item and predict which output changes.

Code Reading Stack

Layer	What to read
Intent	Build a Combined GEO and Technical Scorecard: connect the surface task to the chapter skill, score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit.
Names	Watch these names first: technical, of, geo, concise_answers, schema_aeo. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule; conditions choose a route.
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: technical = 28 # out of 32 | Core: geo_score = sum(1 for v in geo.values() if v) | Evidence: print('top gap:', sorted(k for k, v in geo.items() if not v)[0])

🛠️ Reader to Writer Bridge

1. Read: explain build a combined geo and technical scorecard in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 499.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: write a sibling program with the same skeleton but a new purpose. If your new Python code still has setup, logic, and evidence, use this as the chapter checkpoint before moving to the next topic.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print('top gap:', sorted(k for k, v in geo.items() if not v)[0])`. Then compare punctuation, spacing, and order.

   prediction before proof

2. Explain: point to `technical = 28 # out of 32` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of build a combined geo and technical scorecard and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🔍 From Concept to Code (Python)

Adding the technical score to the GEO/AEO score over a combined 37-point total produces one percentage and grade that balance crawlability and answer-readiness.

```
technical = 28  # out of 32
geo = {'concise_answers': True, 'schema_aeo': True, 'eeat_strong': True,
'fresh': False, 'entities_clear': True}
geo_score = sum(1 for v in geo.values() if v)
geo_total = len(geo)
combined = technical + geo_score
combined_total = 32 + geo_total
pct = round(100 * combined / combined_total)
g = 'A' if pct >= 90 else 'B' if pct >= 80 else 'C' if pct >= 70 else 'D' if
pct >= 60 else 'F'
print('=== GEO + TECHNICAL SCORECARD ===')
print(f'technical:    {technical}/32')
print(f'GEO/AEO:      {geo_score}/{geo_total}')
print(f'combined:     {combined}/{combined_total} ({pct}%) grade {g}')
print('top gap:', sorted(k for k, v in geo.items() if not v)[0])
```

```
output
=== GEO + TECHNICAL SCORECARD ===
technical:    28/32
GEO/AEO:      4/5
combined:     32/37 (86%) grade B
top gap: fresh
```

How it works:

1. A technical score out of thirty-two is given, and a GEO dictionary holds five readiness booleans.
2. geo_score counts the passing GEO signals and geo_total is the dictionary length.
3. The combined score and total add the technical and GEO figures together.
4. A rounded percentage drives an inline chained grade band of A through F.
5. A banner prints the technical, GEO and combined scores, and the sorted first GEO gap.

Try it:

1. Weight the technical and GEO halves rather than adding their raw points equally.
2. List all GEO gaps in priority order instead of only the first one.
3. Track the combined score over time to show progress on both dimensions at once.

4. Push the technical score to 32 and the GEO score to 5 for a grade-A sheet.

⚠ Common Mistakes

- Combining raw scores without a shared denominator would make the percentage meaningless.
- If every GEO signal passed, `sorted(...)[0]` over an empty list would raise; guard for a clean sheet.
- Weighting technical and GEO equally per-point may under-value whichever matters more for the site.

Skills gained combined scorecard | score fusion | grade banding | GEO plus technical

→ Where You'll Use This: A unified technical-plus-GEO scorecard is the deliverable a modern SEO audit should produce, giving stakeholders one health figure that spans both crawlability and answer-engine readiness.

Mental Model

Like a hybrid car's combined rating: it is not enough to score well on the engine alone; the final grade fuses traditional power with the new electric drive.

✅ Chapter Progress Checkpoint

What this chapter added: after 36 mini-projects, GEO, AEO and the End-to-End Technical Audit should feel like a reusable Python move: score answer-extractability and e-e-a-t signals, ai-mode readiness, and run a 32-point technical audit. *chapter checkpoint*

1 Read
Open any project from Projects 465-500 and mark setup, rule, and evidence without looking back at the explanation.

2 Write
Create one sibling program that keeps the same logic but changes the purpose, data, or output wording.
transfer proves learning

3 Explain
Say what this chapter builds on and where it leads: Next, the same habits become your independent review loop: read, trace, modify, test, and explain.

Ready-to-move-on test: if you can predict a project, explain the rule, and safely change one detail, the concept is becoming a writing skill rather than a copied example.

Quiz

9 questions · 4 categories · graded

3 Recall | 2 Meaning | 2 Structure | 2 Transfer

14.1.Q1 Recall MEDIUM Turn back to Project 488 (Write an Hreflang Audit Check). Without running it, what exactly does it print?

14.1.Q2 Meaning MEDIUM Turn back to Project 483 (Write a robots.txt Audit Check). Without running it, what exactly does it print?

14.1.Q3 Structure MEDIUM Turn back to Project 487 (Write a Core Web Vitals Audit Check). Without running it, what exactly does it print?

14.1.Q4 Transfer MEDIUM Turn back to Project 465 (Score a Page's Answer-Extractability Lead). Without running it, what exactly does it print?

14.1.Q5 Recall EASY What is the core difference between classic SEO and GEO/AEO?

A. GEO/AEO ignores keywords entirely

B. GEO/AEO optimises to be the source an answer engine extracts and cites, not just a ranked blue link

C. GEO/AEO only applies to images

D. They are the same thing with different names

14.1.Q6 Meaning MEDIUM An answer engine is choosing which sentence to quote for a query. Which page is most likely to be picked?

A. The longest article on the topic

B. A page whose first paragraph is a tight 30-word direct answer in a quotable shape

C. A page that mentions the keyword the most times

D. A page with the most images

14.1.Q7 Structure EASY Why turn E-E-A-T into a checklist score (e.g. /7) instead of judging it by feel?

A. Because Google publishes the exact formula

B. Because engines infer trust from concrete signals (author bio, credentials, dates, citations, about/contact), so summing those signals makes a fuzzy concept measurable and actionable

C. Because E-E-A-T does not affect ranking

D. Because it removes the need for good content

14.1.Q8 Transfer MEDIUM The 32-point audit returns 28/32 with failures [no_orphans, x_default, cls_ok, fresh]. Why is mapping each failure to a fix the essential last step?

14.1.Q9 Recall MEDIUM A page is fully extractable (concise lead, list, table) but has no named author, no dates, and no outbound citations. What is the likely outcome with answer engines?

Answer key for this chapter follows. Score yourself: 80%+ to advance, otherwise re-read the grammar block and re-take the quiz.

Answer Key – Chapter 14 – GEO, AEO and the End-to-End Technical Audit

14.1.Q1. `hreflang check: PASS | missing-return: [] xdefault: True`

14.1.Q2. `robots.txt check: PASS` then `site-wide block: FAIL`

14.1.Q3. `passing vitals: ['LCP', 'INP']` then `Core Web Vitals check: FAIL`

14.1.Q4. `lead answer words: 27` then `extractable lead: YES`

14.1.Q5. B answer engines synthesise an answer from several pages and cite a few; GEO/AEO is about being one of those cited, extracted sources.

14.1.Q6. B extractability wins: a concise, self-contained lead answer is the easiest passage to lift, regardless of overall length.

14.1.Q7. B the concept is fuzzy but its inputs are concrete page signals, so a checklist turns it into a number you can improve.

14.1.Q8. A list of failing check names is not actionable on its own; mapping each (e.g. fresh → refresh content and update lastmod) turns the score into a remediation plan a person can execute, and the audit can then be re-run to confirm the fix.

14.1.Q9. It is quotable but rarely quoted — extraction is satisfied but trust is not, and a citation needs both. The fix is to add E-E-A-T signals (author bio with credentials, published/updated dates, citations to authoritative sources).

Appendix A – Glossary

@context. the key naming the vocabulary the field names come from — always "https://schema.org" for SEO.

@type. the key naming what the object is, such as Article, Product, FAQPage, BreadcrumbList or LocalBusiness.

301 redirect. a permanent redirect; engines pass ranking signals to the target and swap the old URL for it in the index.

302 redirect. a temporary redirect; engines keep the original URL indexed and transfer little signal to the target.

access log. the server-written diary of every request received, one line per hit, in arrival order; ground truth for what was actually crawled.

AEO. answer engine optimization — structuring content (question headings, concise answers, FAQ/HowTo schema) so answer engines can extract a direct answer.

alt text. the `alt` attribute on an image; describes it for screen readers and crawlers. Missing = fault, empty = decorative, filled = described.

anchor text. the clickable words of a link, read as a description of the target; a healthy profile spreads anchor types instead of repeating one exact-match phrase.

answer-extractability. how easily a machine can lift a self-contained answer: a concise lead, a clean definition, a list, or a table.

cannibalization. when two or more of your own pages target the same keyword and compete, splitting rankings instead of one page winning.

canonical link. a `<link rel="canonical">` naming the master URL so duplicate variants consolidate their ranking signals.

canonical tag. a `<link rel="canonical">` hint naming which URL is the indexing representative for a set of duplicates; ordinary pages self-reference.

changefreq / priority. optional crawl hints: a keyword from always..never, and a 0.0–1.0 decimal (default 0.5) — neither is a ranking signal.

citation. an online mention of a business's NAP, such as a Yelp, Apple Maps, or directory entry.

CLS. Cumulative Layout Shift — a unitless visual-stability score; good ≤ 0.1, poor > 0.25.

Common Log Format. the standard log line layout packing IP, timestamp, request line, status, bytes, referer and User-Agent into one space-separated string.

content freshness. how recently a page was published or updated; flagged stale when older than a chosen audit threshold (e.g. one year).

crawl budget. the number of fetches a bot will make on a site in a window; measured from the log as the bot-hit count and the distinct paths reached.

crawl waste. bot fetches spent on dead ends — 404s, redirects, blocked paths or duplicate parameter URLs — and so stolen from pages that matter.

crawl-delay. a (non-standard) directive asking a bot to wait N seconds between fetches; a per-minute fetch ceiling of 60/N.

crawler. an automated bot that fetches pages and follows their links to discover more URLs.

damping factor. the chance (classically 0.85) the surfer follows a link rather than teleporting; the (1 - d) slice keeps every page above zero and totals at 1.0.

duplicate content. multiple URLs serving identical visible text, detected by grouping pages with the same content hash (e.g. hashlib.sha1).

E-E-A-T. Experience, Expertise, Authoritativeness, Trust — inferred from signals like author bio, credentials, citations, and about/contact pages.

field vs lab data. field data is real users (CrUX); lab data is a single synthetic test on a simulated device.

GEO. generative engine optimization — making a page the content a generative AI engine selects, synthesises from, and cites.

Google Business Profile. a business's Google listing (name, address, phone, hours, categories, photos) whose completeness and verification feed local ranking.

H1 / heading hierarchy. the <h1>–<h6> outline; a sound page has one H1 and steps levels down one at a time with no skips.

hreflang. an annotation mapping each language/region version of a page to its URL, so the engine serves the right version per user.

hreflang cluster. the set of all language alternates for one piece of content; the same block is repeated identically on every page in it.

in-degree. the number of links pointing at a page; high in-degree is where link equity collects.

INP. Interaction to Next Paint — responsiveness from interaction to the next painted frame; good ≤ 200ms, poor > 500ms. Replaced FID in 2024.

inverted index. a map from each term to the set of pages containing it — the structure a query is answered from.

Jaccard similarity. shared shingles over total shingles (intersection / union), a 0-to-1 overlap score; the classic near-duplicate metric.

JSON-LD. JSON for Linked Data; the recommended structured-data format, placed in a <script type="application/ld+json"> tag.

keyword cluster. a set of keywords sharing intent and topic that belong on one pillar page rather than many.

keyword difficulty. a 0–100 estimate of how hard it is to reach page one for a keyword, based on the strength of the pages already ranking.

lastmod. an optional ISO/W3C date (e.g. 2026-01-01) telling crawlers when the page last changed.

LCP. Largest Contentful Paint — time until the largest visible element renders; good ≤ 2.5s, poor > 4.0s.

link equity. the ranking value a link passes; a page splits it evenly across its out-links, so more out-links dilute each share.

loc. the required absolute URL inside each <url>; must be fully-qualified, same-host and (in practice) HTTPS.

local pack. the map plus (usually three) business profiles shown atop a local-intent search.

locale code. an hreflang value: a lowercase ISO 639-1 language, optionally a hyphen and an uppercase ISO 3166-1 alpha-2 region (e.g. `en-GB`), plus the special `x-default`.

long-tail keyword. a longer, more specific query (often 4+ words): low volume, low difficulty, high intent.

longest-match precedence. when several rules match a path, the one whose pattern matches the most characters wins; Allow breaks a length tie.

meta description. a `<meta name="description">` whose text often becomes the results snippet; drives clicks and is clipped near ~155 characters.

NAP. Name, Address, Phone — the core identifying facts that must match across every listing.

NAP consistency. the share of a business's citations whose NAP exactly matches the canonical record.

nofollow. a rel attribute marking a link as a non-vote: it may pass traffic and signal interest but passes no ranking equity.

orphan page. a page no internal link points to, so the crawler may never discover it.

p75. the 75th percentile of real-user field data; the value Google grades a metric against.

PageRank. a page's long-run share of a random surfer's visits, found by iterating the even-split link flow with damping until it converges.

performance budget. a hard cap on transfer size, per-type KB, request count or score that a page may not exceed.

priority score. a single combined number, e.g. volume × intent_weight ÷ difficulty, used to rank a keyword list into action order.

prominence. how well-known and trusted a business is, driven largely by reviews and citation breadth.

redirect chain. a sequence of redirects (A → B → C → page); extra hops add latency and bleed link equity.

redirect loop. a cycle of redirects that never reaches a live page; caught with a seen-set or a hop limit.

referring domain. a distinct site that links to you; the headline backlink metric, since independent domains count far more than repeated links from one site.

render-blocking resource. CSS or synchronous JavaScript in the head that the browser must process before it can paint, delaying LCP.

required property. a field a type must have, or it earns no rich result; recommended properties only enrich an already-valid result.

return tag (reciprocity). the requirement that if page A names B as an alternate, B must name A back, or the relationship is ignored.

rich result. an enhanced search listing (stars, prices, FAQ accordion, breadcrumbs) granted to a page with valid structured data.

robots meta. a `<meta name="robots">` carrying directives like `noindex` (keep out of the index) and `nofollow` (do not pass link credit).

robots.txt. a root-level text file that tells crawlers which paths they may fetch; controls crawling, not indexing, and is obeyed voluntarily.

search intent. the goal behind a query: informational, navigational, commercial, or transactional. Decides what content can rank.

search volume. estimated monthly searches for a keyword — its demand, independent of how hard it is to rank for.

self-reference. the rule that every page must list its own locale among its hreflang alternates, or the cluster is invalid for that page.

shingle. a sliding window of k consecutive tokens; the set of a page's shingles is what near-duplicate detectors compare.

sitemap. an XML file listing the URLs you want crawled, giving the bot a discovery seed independent of internal links.

sitemapindex. a root element whose <sitemap> children point to child sitemaps, used to fan out past the 50,000-URL / 50 MB limit.

soft-404. a page that returns 200 but whose body says it is missing, so engines index nothing of value.

status class. a status code grouped by its leading digit: 2xx success, 3xx redirect, 4xx client error, 5xx server error.

status code. the three-digit integer a server returns for each request; its first digit sets the class (2xx/3xx/4xx/5xx).

structured data. a machine-readable fact sheet attached to a page, stating what its content is in an agreed vocabulary.

technical audit. a repeatable pass running many checks over a site, scoring them out of a total, grading the percentage, and listing fixes.

term frequency. how often a query term appears in a page relative to the page length; an early ranking signal.

thin content. a page with too few unique words or too much boilerplate to satisfy a searcher; a soft 404 (200 status, no real content) is the extreme case.

title tag. the `<title>` in the head; the clickable results headline and the strongest on-page signal, healthiest at ~50-60 characters.

URL normalization. rewriting every variant of a URL into one canonical string (lowercase host, drop fragment, strip tracking params, sort query, trim slash) so duplicates become equal.

urlset. the root element of a normal sitemap, containing one <url> entry per page.

user-agent group. a `User-agent:` line and the Allow/Disallow rules beneath it; a bot obeys one group — its own, or * as fallback.

wildcard. pattern characters * (any run of characters) and trailing $ (end of path) that let one rule target a family of URLs.

x-default. the hreflang value naming the fallback page for users whose locale matches none you serve.

X-Robots-Tag. an HTTP response header carrying indexing directives (noindex, nofollow, noarchive) for files that cannot hold a meta tag, such as PDFs.

Appendix B – Operator & Symbol Reference

The operators that appear throughout the projects, gathered for quick reference.

Operator	Meaning
`+ − * / // % **`	add, subtract, multiply, true-divide, floor-divide, remainder, power
`== != < <= > >=`	equality and ordering comparisons
`and or not`	logical AND, OR, NOT
`in / not in`	membership test in strings, lists, sets, dicts
`= += −= *= /=`	assignment and in-place update
`[a:b:c]`	slice from a up to (not including) b, stepping by c
`f"…{x}…"`	f-string: interpolate the value of x into text
`& | ^ ~ << >>`	bitwise AND, OR, XOR, NOT, shifts