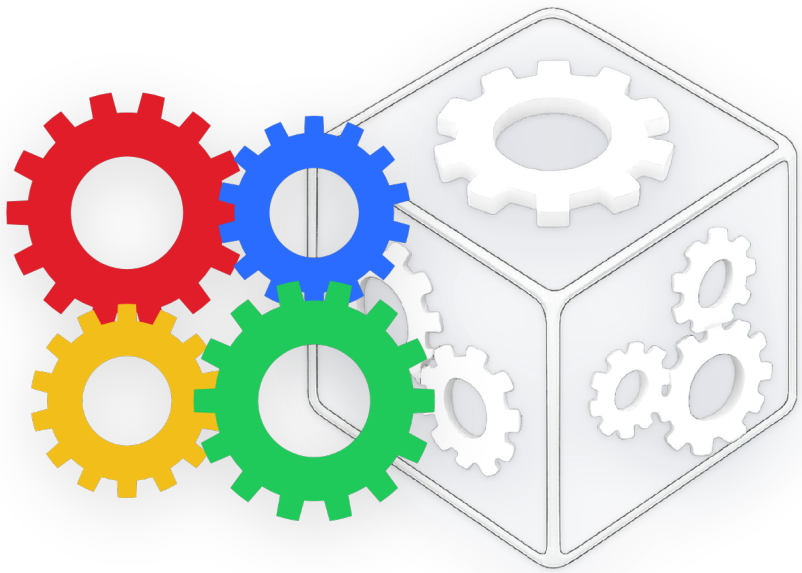


AI ENGINEERING · PROJECTS

AI Engineering Essentials

A Guided AI Engineering Course with Executed Practice Projects

2026 Edition · Luke McLaughlin

AI Engineering Essentials

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 AI engineering into a guided course: 16 chapters, full concept lessons, worked traces, visual models, reader-to-writer bridges, checkpoints, quizzes and executed practice projects.

The book keeps the familiar code-and-output rhythm, but every project is framed as a transferable AI habit: read the data shape, name the transformation, predict the metric or output, then change one detail safely.

Work through the chapters in order if you want a complete path from NumPy and data preparation to metrics, classical machine learning, neural-network maths, PyTorch, NLP, RAG, scaling and MLOps.

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 follows the same rhythm. Look for these signposts:

- 💡 ***The Idea Behind It*** – each chapter opens with *why* its concept matters – the thinking behind the code.
- ***Project N → code + output*** – the complete, runnable program together with the exact output to check against.
- ⚠ ***Common Mistakes*** – the traps most learners fall into – and how to sidestep them.
- ***Try it*** – small modifications that turn the example into your own experiment.
- 📝 ***Quiz*** – a few questions at the end of every chapter. Full answers follow in the chapter's Answer Key.

Work through the parts in order – each one assumes the ones before it. Take the quizzes seriously; the goal is not to finish fast but to *understand*.

Contents

Part I – Beginner Foundations

- Chapter 1 – NumPy Fundamentals
- Chapter 2 – NumPy Math, Broadcasting & Linear Algebra
- Chapter 3 – pandas Essentials
- Chapter 4 – Data Preparation & Feature Engineering
- Chapter 5 – Evaluation Metrics from Scratch

Part II – Intermediate Skills

- Chapter 6 – Classical ML with scikit-learn
- Chapter 7 – Regression & Classification from Scratch
- Chapter 8 – Optimization & Gradient Descent
- Chapter 9 – Neural Networks from Scratch
- Chapter 10 – PyTorch Fundamentals

Part III – Advanced Mastery

- Chapter 11 – Vision: Images as Arrays
- Chapter 12 – NLP & Text Processing
- Chapter 13 – Embeddings & Similarity Search
- Chapter 14 – Transformers & Attention from Scratch
- Chapter 15 – LLM Engineering, Prompts & RAG
- Chapter 16 – Scaling, Compute, Cost & MLOps



PART I

Beginner Foundations

Start from absolute zero. These projects build the core habits of a programmer one small, runnable step at a time.

Chapter 1 – NumPy Fundamentals · Chapter 2 – NumPy Math, Broadcasting & Linear Algebra · Chapter 3 – pandas Essentials · Chapter 4 – Data Preparation & Feature Engineering · Chapter 5 – Evaluation Metrics from Scratch

Chapter 1 · NumPy Fundamentals

Arrays, dtypes, indexing, masking — the substrate of all AI work.

Chapter Skill Map

What this chapter adds: In NumPy Fundamentals, the focus is arrays, dtypes, indexing, masking — the substrate of all ai work. You practise it across 39 small Python decisions, building a repeatable reading-and-writing habit rather than racing 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, NumPy Math, Broadcasting & Linear Algebra, reuses this chapter's discipline with a new kind of problem.

1 **Projects 1-13**	2 **Projects 14-26**	3 **Projects 27-39**
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

NumPy Fundamentals: what problem does this solve for a working programmer?

name the move before typing

NumPy arrays are the substrate of AI work: shape, dtype and indexing determine what every later model can see.

This chapter's focus is: Arrays, dtypes, indexing, masking — the substrate of all AI work. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```python
import numpy as np

values = np.array([2, 4, 6])
print(values.shape)
print(values[values > 3])
```

```
output
(3,)
[4 6]
```

The boolean mask values > 3 selects only the entries that satisfy the condition. *predict first, verify second*

ARRAY

A typed block of values with a shape.

DTYPE

The numeric representation, such as int64 or float32.

INDEXING

Selecting values by position, slice or boolean mask.

SHAPE

The dimensions of an array, such as (rows, columns).

Beginner model

At beginner level, NumPy Fundamentals should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Deep-learning tensors are NumPy's idea scaled to accelerators and automatic differentiation.

The edge case to respect

Shape mismatches are the first serious AI bug: the numbers may be right but arranged wrong.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Create a 2-D array and select one row, one column and one boolean mask.

Example 1

Trace the opening NumPy Fundamentals example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read NumPy Fundamentals as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Create a 2-D array and select one row, one column and one boolean mask.

⚠ Common Mistakes

- Copying the surface syntax for NumPy Fundamentals without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Shape mismatches are the first serious AI bug: the numbers may be right but arranged wrong.
- Trusting the output before predicting it by hand.

This chapter contains 39 hands-on mini-projects (Projects 1–39). Type each one out, run it, and experiment.

Project 1 Summarise arange 1..4

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..4 into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example. *first anchor* Project 2 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

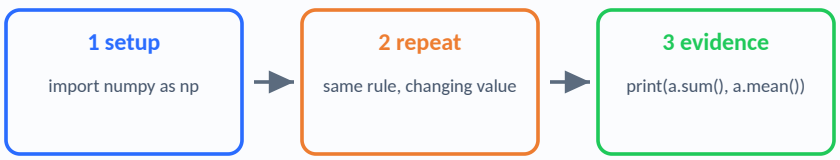
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 2 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..4 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 5)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 5)`

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. 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	Summarise arange 1..4: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 5) | Evidence: print(a.sum(), a.mean())

Reader to Writer Bridge

1. Read: explain summarise arange 1..4 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..4 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 5)
print(a.sum(), a.mean())
```

```
output
10 2.5
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.
- Calling .sum() when you needed the axis-wise version with axis=.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 2 Summarise arange 1..5

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..5 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 1, where you practised summarise arange 1..4, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 3 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

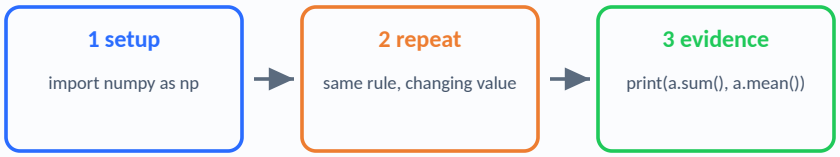
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 1: summarise arange 1..4. Project 3 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 6)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 6)`

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. 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	Summarise arange 1..5: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 6) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..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 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..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)

np.arange builds a numeric range; sum/mean reduce it.

```
import numpy as np
```

```
a = np.arange(1, 6)
print(a.sum(), a.mean())
```

```
output
15 3.0
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).
- Mixing dtypes silently: ints become floats and == comparisons drift.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 3 Summarise arange 1..6

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..6 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 2, where you practised summarise arange 1..5, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 4 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

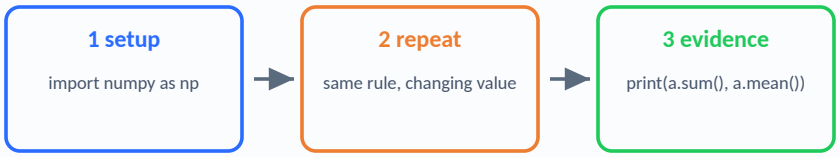
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 2: summarise arange 1..5. Project 4 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..6 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 7)`

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. 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	Summarise arange 1..6: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 7) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..6 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..6 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 7)
print(a.sum(), a.mean())
```

```
output
21 3.5
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).
- Forgetting that arange(a, b) excludes b.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 4 Summarise arange 1..7

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..7 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.

- It builds on Project 3, where you practised summarise arange 1..6, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 5 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

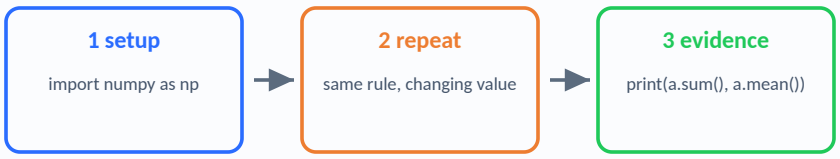
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 3: summarise arange 1..6. Project 5 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..7 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 8)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 8)`

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. 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	Summarise arange 1..7: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 8) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..7 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..7 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 8)
print(a.sum(), a.mean())
```

```
output
28 4.0
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Printing huge arrays instead of checking .shape and .dtype first.
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 5 Summarise arange 1..8

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..8 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 4, where you practised summarise arange 1..7, and asks you to reuse the same reasoning pattern. reuse the pattern Project 6 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 4: summarise arange 1..7. Project 6 changes the surface problem while preserving the underlying method. progress check

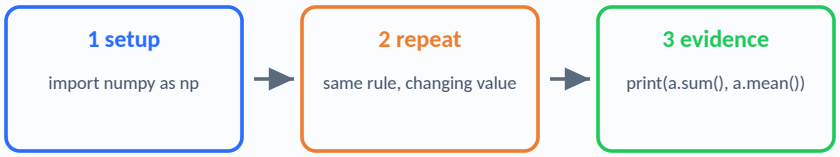
Reading target: summarise arange 1..8 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 9)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 9)`

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. 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	Summarise arange 1..8: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 9) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..8 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import numpy as np` and say why 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 arange 1..8 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 9)
print(a.sum(), a.mean())
```

```
output
36 4.5
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Assuming reshape copies — it usually returns a view of the same data.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 6 Summarise arange 1..9

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..9 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 5, where you practised summarise arange 1..8, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 7 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

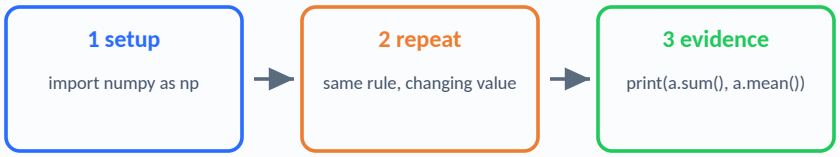
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 5: summarise arange 1..8. Project 7 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..9 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 10)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 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. 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	Summarise arange 1..9: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 10) | Evidence: print(a.sum(), a.mean())

Reader to Writer Bridge

1. Read: explain summarise arange 1..9 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..9 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 10)
print(a.sum(), a.mean())
```

```
output
45 5.0
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Calling .sum() when you needed the axis-wise version with axis=.
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 7 Summarise arange 1..10

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..10 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 6, where you practised summarise arange 1..9, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 8 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

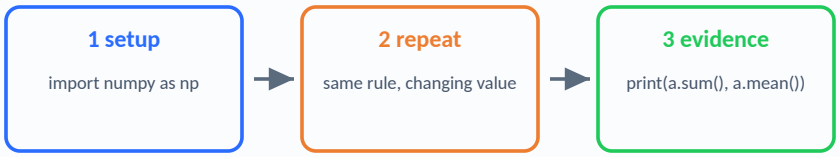
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 6: summarise arange 1..9. Project 8 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..10 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 11)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 11)`

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. 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	Summarise arange 1..10: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 11) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..10 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..10 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```
import numpy as np
```

```
a = np.arange(1, 11)
print(a.sum(), a.mean())
```

```
output
55 5.5
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Mixing dtypes silently: ints become floats and == comparisons drift.
- Printing huge arrays instead of checking .shape and .dtype first.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 8 Sort an Array (set 2)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 7, where you practised summarise arange 1..10, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 9 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 7: summarise arange 1..10. Project 9 changes the surface problem while preserving the underlying method.

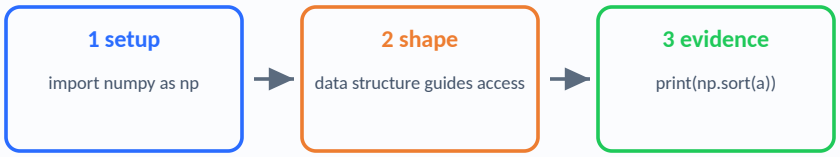
progress check

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([8, 19, 30, 41, 52])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([8, 19, 30, 41, 52])`

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. 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	Sort an Array (set 2): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([8, 19, 30, 41, 52]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([8, 19, 30, 41, 52])
print(np.sort(a))
```

```
output
[ 8 19 30 41 52]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Forgetting that arange(a, b) excludes b.
- Assuming reshape copies — it usually returns a view of the same data.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 9 Sort an Array (set 7)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 8, where you practised sort an array, and asks you to reuse the same reasoning pattern.

reuse the pattern Project 10 will change the surface task again, so treat this one as a checkpoint:

predict the output before you look ahead. *predict first*

Skill Roadmap

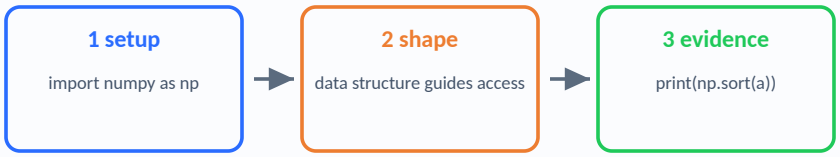
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 8: sort an array. Project 10 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([43, 54, 65, 76, 87])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([43, 54, 65, 76, 87])`

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. 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	Sort an Array (set 7): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([43, 54, 65, 76, 87]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```
import numpy as np

a = np.array([43, 54, 65, 76, 87])
print(np.sort(a))
```

```
output
[43 54 65 76 87]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.
- Calling .sum() when you needed the axis-wise version with axis=.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 10 Sort an Array (set 12)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 9, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 11 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 9: sort an array. Project 11 changes the surface problem while preserving the underlying method. *progress check*

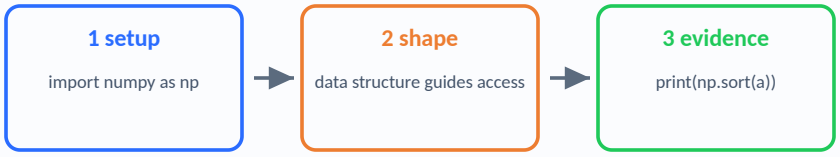
Reading target: sort an array is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.

- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([78, 89, 10, 21, 32])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([78, 89, 10, 21, 32])`

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. 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	Sort an Array (set 12): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([78, 89, 10, 21, 32]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([78, 89, 10, 21, 32])
print(np.sort(a))
```

```
output
[10 21 32 78 89]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*

2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).
- Mixing dtypes silently: ints become floats and == comparisons drift.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 11 Sort an Array (set 17)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.

- Read the program as input or setup, transformation, then output. three-pass reading
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 10, where you practised sort an array, and asks you to reuse the same reasoning pattern. reuse the pattern Project 12 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

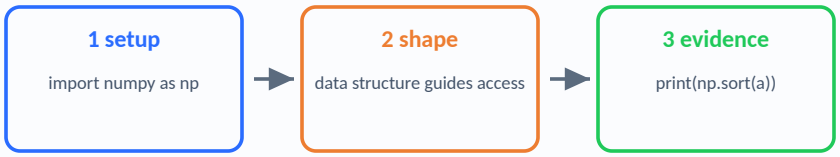
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 10: sort an array. Project 12 changes the surface problem while preserving the underlying method. progress check

Reading target: sort an array is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.

Move	Code to inspect	Why it matters
2. Apply the rule	`a = np.array([23, 34, 45, 56, 67])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([23, 34, 45, 56, 67])`

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. 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	Sort an Array (set 17): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([23, 34, 45, 56, 67]) | Evidence: print(np.sort(a))

Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([23, 34, 45, 56, 67])
print(np.sort(a))
```

```
output
[23 34 45 56 67]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).
- Forgetting that arange(a, b) excludes b.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 12 Sort an Array (set 22)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 11, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 13 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 11: sort an array. Project 13 changes the surface problem while preserving the underlying method. *progress check*

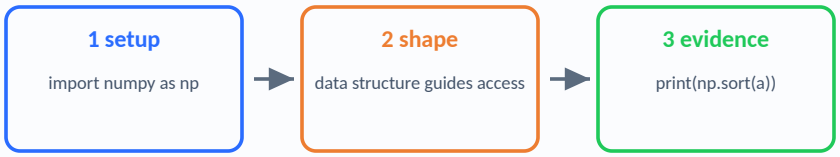
Reading target: sort an array is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.

- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([58, 69, 80, 1, 12])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([58, 69, 80, 1, 12])`

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. 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	Sort an Array (set 22): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([58, 69, 80, 1, 12]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([58, 69, 80, 1, 12])
print(np.sort(a))
```

```
output
[ 1 12 58 69 80]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Printing huge arrays instead of checking .shape and .dtype first.
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 13 Sort an Array (set 27)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 12, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 14 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

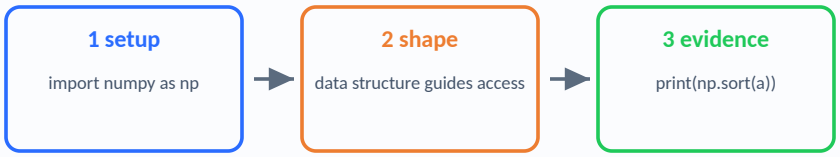
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 12: sort an array. Project 14 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([3, 14, 25, 36, 47])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([3, 14, 25, 36, 47])`

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. 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	Sort an Array (set 27): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([3, 14, 25, 36, 47]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([3, 14, 25, 36, 47])
print(np.sort(a))
```

```
output
[ 3 14 25 36 47]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Assuming reshape copies — it usually returns a view of the same data.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 14 Sort an Array (set 32)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 13, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 15 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

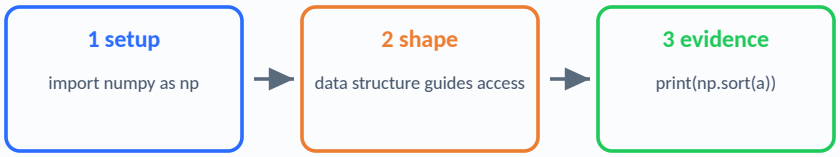
Skill step: moves you from recognition toward independent tracing. It builds on Project 13: sort an array. Project 15 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([38, 49, 60, 71, 82])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([38, 49, 60, 71, 82])`

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. 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	Sort an Array (set 32): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([38, 49, 60, 71, 82]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([38, 49, 60, 71, 82])
print(np.sort(a))
```

```
output
[38 49 60 71 82]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Calling .sum() when you needed the axis-wise version with axis=.
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 15 Boolean Mask: Above Mean (set 4)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 14, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 16 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

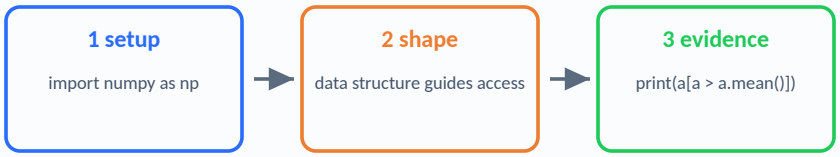
Skill step: moves you from recognition toward independent tracing. It builds on Project 14: sort an array. Project 16 changes the surface problem while preserving the underlying method. *progress check*

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([9, 14, 19, 24, 29, 34])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([9, 14, 19, 24, 29, 34])`

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. 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	Boolean Mask: Above Mean (set 4): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([9, 14, 19, 24, 29, 34]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([9, 14, 19, 24, 29, 34])
print(a[a > a.mean()])
```

```
output
[24 29 34]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Mixing dtypes silently: ints become floats and == comparisons drift.
- Printing huge arrays instead of checking .shape and .dtype first.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 16 Boolean Mask: Above Mean (set 9)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 15, where you practised boolean mask: above mean, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 17 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 15: boolean mask: above mean. Project 17 changes the surface problem while preserving the underlying method.

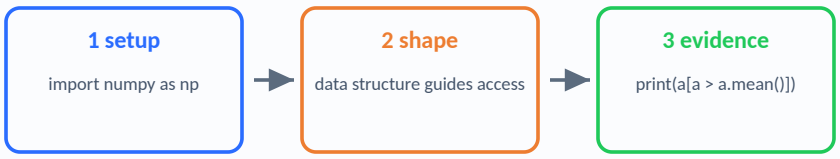
progress check

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([24, 29, 34, 39, 44, 49])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([24, 29, 34, 39, 44, 49])`

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. 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	Boolean Mask: Above Mean (set 9): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([24, 29, 34, 39, 44, 49]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([24, 29, 34, 39, 44, 49])
print(a[a > a.mean()])
```

```
output
[39 44 49]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Forgetting that arange(a, b) excludes b.
- Assuming reshape copies — it usually returns a view of the same data.

Skills gained choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 17 Boolean Mask: Above Mean (set 14)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 16, where you practised boolean mask: above mean, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 18 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 16: boolean mask: above mean. Project 18 changes the surface problem while preserving the underlying method.

progress check

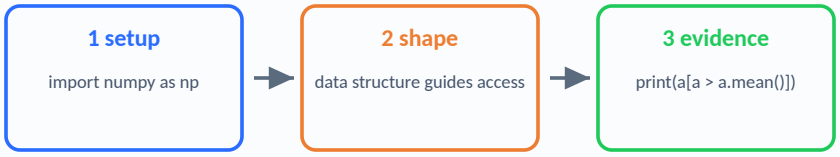
Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

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

- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([39, 44, 49, 4, 9, 14])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([39, 44, 49, 4, 9, 14])`

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. 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	Boolean Mask: Above Mean (set 14): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([39, 44, 49, 4, 9, 14]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```
import numpy as np
```

```
a = np.array([39, 44, 49, 4, 9, 14])
print(a[a > a.mean()])
```

```
output
[39 44 49]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.
- Calling .sum() when you needed the axis-wise version with axis=.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 18 Boolean Mask: Above Mean (set 19)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 17, where you practised boolean mask: above mean, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 19 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 17: boolean mask: above mean. Project 19 changes the surface problem while preserving the underlying method.

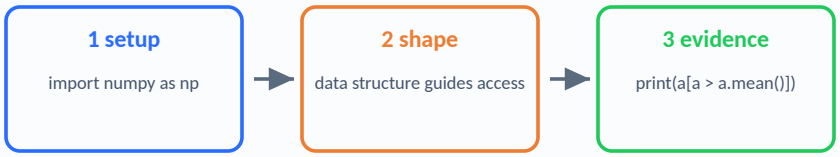
progress check

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([4, 9, 14, 19, 24, 29])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([4, 9, 14, 19, 24, 29])`

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. 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	Boolean Mask: Above Mean (set 19): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([4, 9, 14, 19, 24, 29]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([4, 9, 14, 19, 24, 29])
print(a[a > a.mean()])
```

```
output
[19 24 29]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).
- Mixing dtypes silently: ints become floats and == comparisons drift.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 19 Boolean Mask: Above Mean (set 24)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 18, where you practised boolean mask: above mean, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 20 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 18: boolean mask: above mean. Project 20 changes the surface problem while preserving the underlying method.

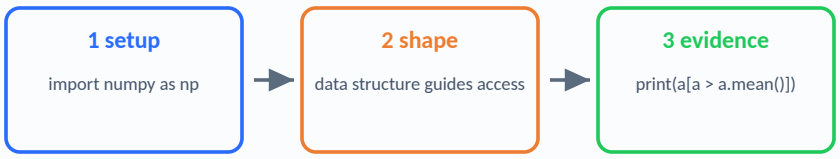
progress check

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([19, 24, 29, 34, 39, 44])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([19, 24, 29, 34, 39, 44])`

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. 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	Boolean Mask: Above Mean (set 24): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([19, 24, 29, 34, 39, 44]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([19, 24, 29, 34, 39, 44])
print(a[a > a.mean()])
```

```
output
[34 39 44]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).
- Forgetting that arange(a, b) excludes b.

Skills gained choosing and reading data structures predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 20 Boolean Mask: Above Mean (set 29)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 19, where you practised boolean mask: above mean, and asks you to reuse the same reasoning pattern. reuse the pattern Project 21 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 19: boolean mask: above mean. Project 21 changes the surface problem while preserving the underlying method.

progress check

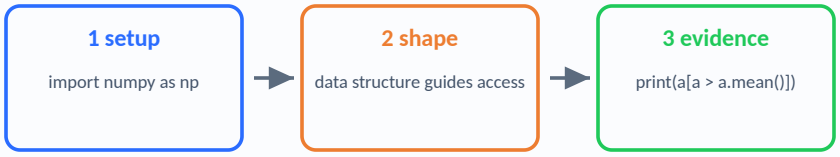
Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

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

- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([34, 39, 44, 49, 4, 9])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([34, 39, 44, 49, 4, 9])`

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. 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	Boolean Mask: Above Mean (set 29): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([34, 39, 44, 49, 4, 9]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```
import numpy as np
```

```
a = np.array([34, 39, 44, 49, 4, 9])
print(a[a > a.mean()])
```

```
output
[34 39 44 49]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Printing huge arrays instead of checking .shape and .dtype first.
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 21 Boolean Mask: Above Mean (set 34)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 20, where you practised boolean mask: above mean, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 22 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 20: boolean mask: above mean. Project 22 changes the surface problem while preserving the underlying method.

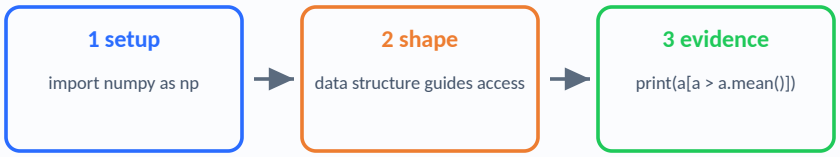
progress check

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([49, 4, 9, 14, 19, 24])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([49, 4, 9, 14, 19, 24])`

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. 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	Boolean Mask: Above Mean (set 34): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([49, 4, 9, 14, 19, 24]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([49, 4, 9, 14, 19, 24])
print(a[a > a.mean()])
```

```
output
[49 24]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Assuming reshape copies — it usually returns a view of the same data.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 22 Column Sums of a 7x7 Matrix (set 5)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 21, where you practised boolean mask: above mean, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 23 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 21: boolean mask: above mean. Project 23 changes the surface problem while preserving the underlying method.

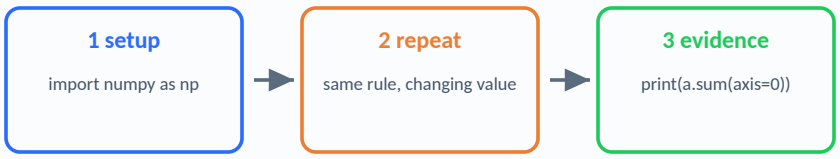
progress check

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 5): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Calling .sum() when you needed the axis-wise version with axis=.
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 23 Column Sums of a 7x7 Matrix (set 10)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 22, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 24 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 22: column sums of a 7x7 matrix. Project 24 changes the surface problem while preserving the underlying method.

progress check

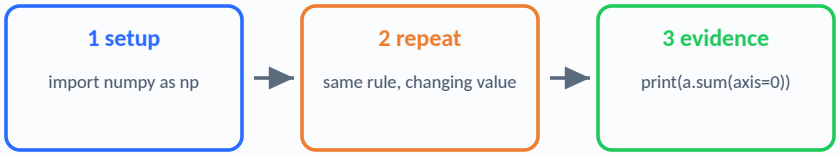
Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

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

- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 10): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Mixing dtypes silently: ints become floats and == comparisons drift.
- Printing huge arrays instead of checking .shape and .dtype first.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 24 Column Sums of a 7x7 Matrix (set 15)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 23, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 25 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 23: column sums of a 7x7 matrix. Project 25 changes the surface problem while preserving the underlying method.

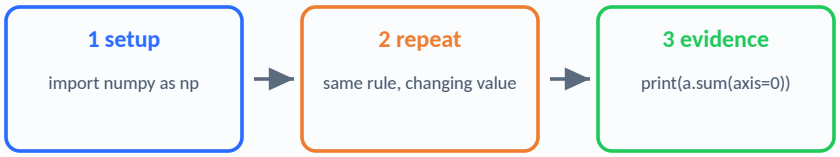
progress check

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 15): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Forgetting that arange(a, b) excludes b.
- Assuming reshape copies — it usually returns a view of the same data.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 25 Column Sums of a 7x7 Matrix (set 20)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 24, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 26 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 24: column sums of a 7x7 matrix. Project 26 changes the surface problem while preserving the underlying method.

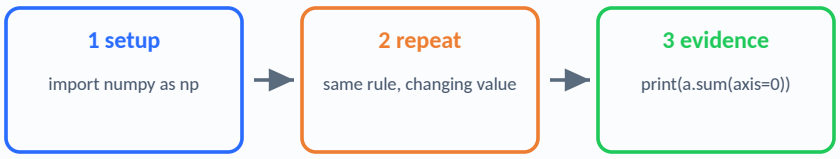
progress check

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 20): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.
- Calling .sum() when you needed the axis-wise version with axis=.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 26 Column Sums of a 7x7 Matrix (set 25)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 25, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 27 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 25: column sums of a 7x7 matrix. Project 27 changes the surface problem while preserving the underlying method.

progress check

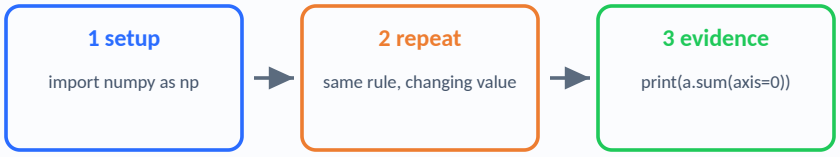
Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

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

- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 25): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).
- Mixing dtypes silently: ints become floats and == comparisons drift.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 27 Column Sums of a 7x7 Matrix (set 30)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 26, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 28 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

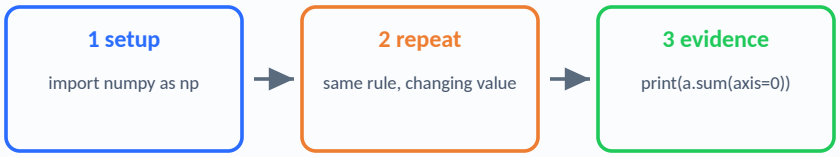
Skill step: pushes the same idea into a less automatic situation. It builds on Project 26: column sums of a 7x7 matrix. Project 28 changes the surface problem while preserving the underlying method. *progress check*

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 30): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).
- Forgetting that arange(a, b) excludes b.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 28 Column Sums of a 7x7 Matrix (set 35)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.

- It builds on Project 27, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 29 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

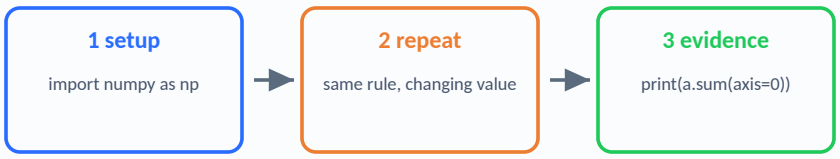
Skill step: pushes the same idea into a less automatic situation. It builds on Project 27: column sums of a 7x7 matrix. Project 29 changes the surface problem while preserving the underlying method. *progress check*

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 35): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Printing huge arrays instead of checking .shape and .dtype first.
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 29 Scale arange 1..6 by 4

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..6 by 4 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 28, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 30 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 28: column sums of a 7x7 matrix. Project 30 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..6 by 4 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

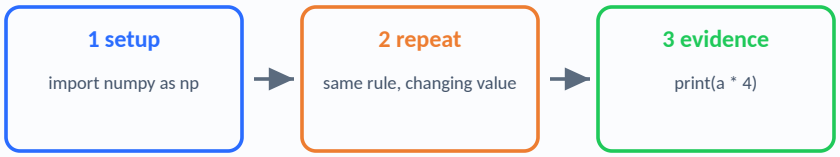
🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a * 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 4)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 7)`

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. 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	Scale arange 1..6 by 4: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 7) | Evidence: print(a * 4)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..6 by 4 in one sentence before touching the 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(a * 4)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..6 by 4 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```python
import numpy as np

a = np.arange(1, 7)
print(a * 4)
```

```
output
[ 4  8 12 16 20 24]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 4)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Assuming reshape copies — it usually returns a view of the same data.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 30 Scale arange 1..7 by 9

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..7 by 9 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 29, where you practised scale arange 1..6 by 4, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 31 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

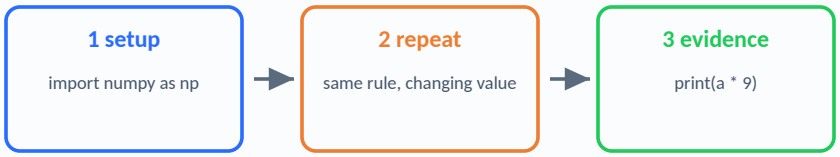
Skill step: pushes the same idea into a less automatic situation. It builds on Project 29: scale arange 1..6 by 4. Project 31 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..7 by 9 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a * 9)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 8)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 9)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 8)`

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. 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	Scale arange 1..7 by 9: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 8) | Evidence: print(a * 9)

Reader to Writer Bridge

1. Read: explain scale arange 1..7 by 9 in one sentence before touching the 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, Project 31 should feel like a variation, 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 * 9)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..7 by 9 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 8)
print(a * 9)
```

```
output
[ 9 18 27 36 45 54 63]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 9)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Calling .sum() when you needed the axis-wise version with axis=.
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 31 Scale arange 1..4 by 5

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..4 by 5 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 30, where you practised scale arange 1..7 by 9, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 32 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 30: scale arange 1..7 by 9. Project 32 changes the surface problem while preserving the underlying method. *progress check*

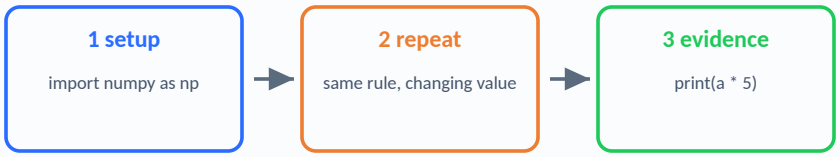
Reading target: scale arange 1..4 by 5 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a * 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 5)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 5)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 5)`

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. 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	Scale arange 1..4 by 5: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 5) | Evidence: print(a * 5)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..4 by 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 30.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: 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(a * 5)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..4 by 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)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np
```

```
a = np.arange(1, 5)
print(a * 5)
```

```
output
[ 5 10 15 20]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 5)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Mixing dtypes silently: ints become floats and == comparisons drift.
- Printing huge arrays instead of checking .shape and .dtype first.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 32 Scale arange 1..5 by 10

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..5 by 10 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 31, where you practised scale arange 1..4 by 5, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 33 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

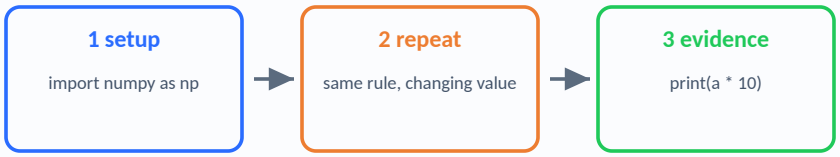
Skill step: pushes the same idea into a less automatic situation. It builds on Project 31: scale arange 1..4 by 5. Project 33 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..5 by 10 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a * 10)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 6)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a * 10)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 6)`

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. 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	Scale arange 1..5 by 10: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 6) | Evidence: print(a * 10)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..5 by 10 in one sentence before touching the 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(a * 10)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..5 by 10 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```python
import numpy as np

a = np.arange(1, 6)
print(a * 10)
```

```
output
[10 20 30 40 50]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a * 10)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Forgetting that arange(a, b) excludes b.
- Assuming reshape copies — it usually returns a view of the same data.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 33 Scale arange 1..6 by 6

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..6 by 6 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.

- It builds on Project 32, where you practised scale arange 1..5 by 10, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 34 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

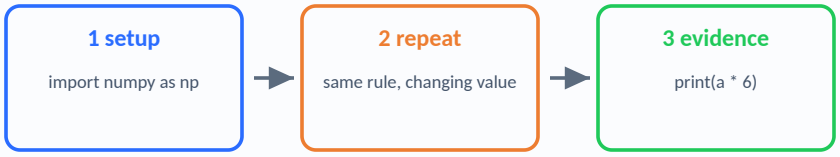
Skill step: pushes the same idea into a less automatic situation. It builds on Project 32: scale arange 1..5 by 10. Project 34 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..6 by 6 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a * 6)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a * 6)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 7)`

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. 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	Scale arange 1..6 by 6: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 7) | Evidence: print(a * 6)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..6 by 6 in one sentence before touching the 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(a * 6)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..6 by 6 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 7)
print(a * 6)
```

```
output
[ 6 12 18 24 30 36]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 6)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.
- Calling .sum() when you needed the axis-wise version with axis=.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 34 Scale arange 1..7 by 2

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..7 by 2 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project pushes the same idea into a less automatic situation.
- It builds on Project 33, where you practised scale arange 1..6 by 6, and asks you to reuse the same reasoning pattern. reuse the pattern Project 35 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 33: scale arange 1..6 by 6. Project 35 changes the surface problem while preserving the underlying method. progress check

Reading target: scale arange 1..7 by 2 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

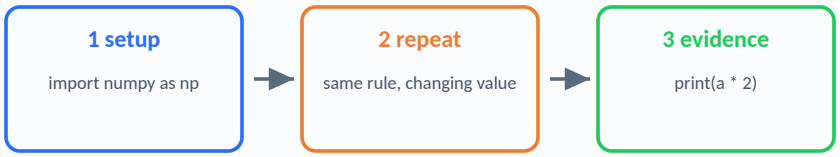
Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a * 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 8)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 2)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 8)`

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. 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	Scale arange 1..7 by 2: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 8) | Evidence: print(a * 2)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..7 by 2 in one sentence before touching the 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(a * 2)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..7 by 2 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```python
import numpy as np

a = np.arange(1, 8)
print(a * 2)
```

```
output
[ 2  4  6  8 10 12 14]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 2)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).
- Mixing dtypes silently: ints become floats and == comparisons drift.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 35 Scale arange 1..4 by 7

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..4 by 7 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 34, where you practised scale arange 1..7 by 2, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 36 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

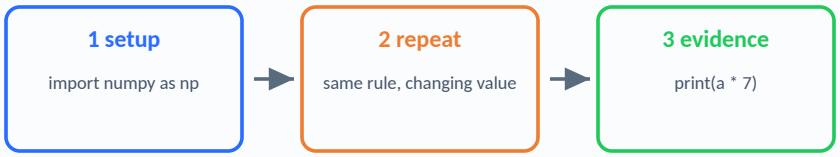
Skill step: pushes the same idea into a less automatic situation. It builds on Project 34: scale arange 1..7 by 2. Project 36 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..4 by 7 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a * 7)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 5)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 7)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 5)`

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. 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	Scale arange 1..4 by 7: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 5) | Evidence: print(a * 7)

Reader to Writer Bridge

1. Read: explain scale arange 1..4 by 7 in one sentence before touching the 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(a * 7)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..4 by 7 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 5)
print(a * 7)
```

```
output
[ 7 14 21 28]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 7)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).
- Forgetting that arange(a, b) excludes b.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 36 Summarise arange 1..11

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..11 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 35, where you practised scale arange 1..4 by 7, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 37 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

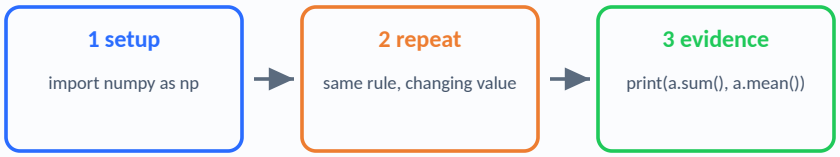
Skill step: pushes the same idea into a less automatic situation. It builds on Project 35: scale arange 1..4 by 7. Project 37 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..11 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 12)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 12)`

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. 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	Summarise arange 1..11: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 12) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..11 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..11 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```
import numpy as np
```

```
a = np.arange(1, 12)
print(a.sum(), a.mean())
```

```
output
66 6.0
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Printing huge arrays instead of checking .shape and .dtype first.
- Expecting a Python list to behave like an array: [1,2] * 2 repeats, it does not scale.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 37 Sort an Array (set 37)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 36, where you practised summarise arange 1..11, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 38 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

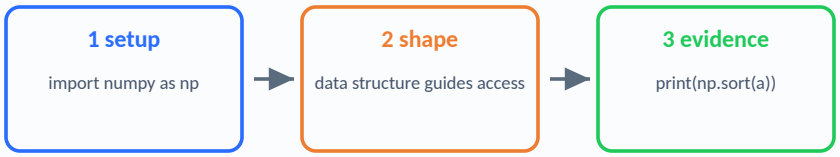
Skill step: pushes the same idea into a less automatic situation. It builds on Project 36: summarise arange 1..11. Project 38 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([73, 84, 5, 16, 27])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([73, 84, 5, 16, 27])`

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. 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	Sort an Array (set 37): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([73, 84, 5, 16, 27]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([73, 84, 5, 16, 27])
print(np.sort(a))
```

```
output
[ 5 16 27 73 84]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Assuming reshape copies — it usually returns a view of the same data.
- Confusing axis=0 (down the rows) with axis=1 (across the columns).

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 38 Scale arange 1..5 by 3

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..5 by 3 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 37, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 39 will change the surface task again, so treat this one as a

checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 37: sort an array. Project 39 changes the surface problem while preserving the underlying method. *progress check*

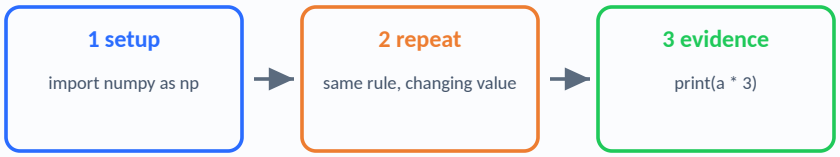
Reading target: scale arange 1..5 by 3 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.
- Evidence: `print(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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.arange(1, 6)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 3)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 6)`

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. 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	Scale arange 1..5 by 3: connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 6) | Evidence: print(a * 3)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..5 by 3 in one sentence before touching the 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(a * 3)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of scale arange 1..5 by 3 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 6)
print(a * 3)
```

```
output
[ 3  6  9 12 15]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 3)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Calling .sum() when you needed the axis-wise version with axis=.
- Reshaping to a size that does not match: 12 elements cannot become (3, 5).

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 39 Boolean Mask: Above Mean (set 39)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 38, where you practised scale arange 1..5 by 3, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 38: scale arange 1..5 by 3. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

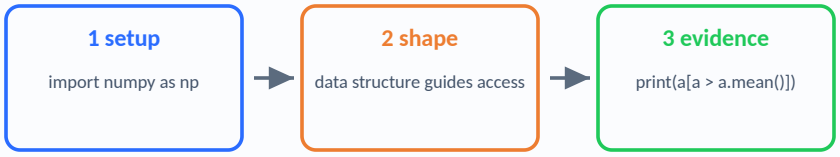
Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: arrays, dtypes, indexing, masking — the substrate of all ai work.

- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: arrays, dtypes, indexing, masking — the substrate of all ai work.
2. Apply the rule	`a = np.array([14, 19, 24, 29, 34, 39])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Fundamentals becomes practical when you can build the syntax 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 = np.array([14, 19, 24, 29, 34, 39])`

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. 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	Boolean Mask: Above Mean (set 39): connect the surface task to the chapter skill, arrays, dtypes, indexing, masking — the substrate of all ai work.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([14, 19, 24, 29, 34, 39]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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, 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([14, 19, 24, 29, 34, 39])
print(a[a > a.mean()])
```

```
output
[29 34 39]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Mixing dtypes silently: ints become floats and == comparisons drift.
- Printing huge arrays instead of checking .shape and .dtype first.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 39 mini-projects, NumPy Fundamentals should feel like a reusable Python move: arrays, dtypes, indexing, masking — the substrate of all ai work. *chapter checkpoint*

1 Read
Open any project from Projects 1-39 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 (NumPy Math, Broadcasting & Linear Algebra) 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

Recall · Meaning · Structure · Transfer

1.1.Q1 Recall Bug hunt: `np.arange(1, 10).reshape(3, 3)` works but `np.arange(10).reshape(3, 3)` throws. Why?

1.1.Q2 Meaning Bug hunt: `m.sum(axis=1)` was expected to sum each column but returned 3 values for a (3, 4) matrix. Which axis sums columns?

1.1.Q3 Structure What is the main problem this chapter's topic solves?

- A It gives a named, repeatable way to handle the chapter's programming situation.
- B It removes the need to test code.
- C It makes every program shorter.
- D It only matters for memorising syntax.

1.1.Q4 Transfer What should you identify before running a project?

- A The setup, the rule or transformation, and the evidence line.
- B Only the final answer.
- C Only the imports.
- D Only the file name.

1.1.Q5 Recall What is the most important edge case in NumPy Fundamentals?

1.1.Q6 Transfer What proves the skill is transferable?

Answer Key – Chapter 1 – NumPy Fundamentals

1.1.Q1. arange(10) has 10 elements; 3×3 needs exactly 9 — reshape never drops or invents data.

1.1.Q2. axis=0 — it collapses the row dimension, leaving one sum per column (4 values).

1.1.Q3. A — NumPy arrays are the substrate of AI work: shape, dtype and indexing determine what every later model can see.

1.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

1.1.Q5. Shape mismatches are the first serious AI bug: the numbers may be right but arranged wrong.

1.1.Q6. Create a 2-D array and select one row, one column and one boolean mask.

Chapter 2 · NumPy Math, Broadcasting & Linear Algebra

Vectorised math, broadcasting rules, matrix operations.

Chapter Skill Map

What this chapter adds: In NumPy Math, Broadcasting & Linear Algebra, the focus is vectorised math, broadcasting rules, matrix operations. You practise it across 5 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 2 assumes the reading habits from Chapter 1: NumPy Fundamentals. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 3, pandas Essentials, reuses this chapter's discipline with a new kind of problem.

1 **Project 40** Recognise the pattern and read the code slowly. *name it first*	2 **Project 41** Trace similar programs with less prompting and explain what changed.	3 **Projects 42-44** Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

NumPy Math, Broadcasting & Linear Algebra: what problem does this solve for a working programmer? *name the move before typing*

Broadcasting and linear algebra turn many scalar operations into one readable vector expression.

This chapter's focus is: Vectorised math, broadcasting rules, matrix operations. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
import numpy as np

x = np.array([[1, 2], [3, 4]])
bias = np.array([10, 100])
print(x + bias)
print(x @ np.array([1, 1]))
```

```
output
[[ 11 102]
 [ 13 104]]
[3 7]
```

bias has shape (2,), so NumPy adds it to each row of x. *predict first, verify second*

BROADCASTING

NumPy's rule for stretching compatible shapes without copying by hand.

VECTORISATION

Applying one operation across many values at once.

DOT PRODUCT

A sum of pairwise products, central to similarity and neural layers.

MATRIX MULTIPLICATION

The @ operation that composes linear transformations.

Beginner model

At beginner level, NumPy Math, Broadcasting & Linear Algebra should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Modern AI is mostly linear algebra plus nonlinearities, repeated at scale.

The edge case to respect

Broadcasting bugs are quiet when a shape is compatible but semantically wrong.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Change bias to a column vector and explain the different result.

Example 1

Trace the opening NumPy Math, Broadcasting & Linear Algebra example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read NumPy Math, Broadcasting & Linear Algebra as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Change bias to a column vector and explain the different result.

⚠ Common Mistakes

- Copying the surface syntax for NumPy Math, Broadcasting & Linear Algebra without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Broadcasting bugs are quiet when a shape is compatible but semantically wrong.
- Trusting the output before predicting it by hand.

This chapter contains 5 hands-on mini-projects (Projects 40–44). Type each one out, run it, and experiment.

Project 40 Column Sums of a 7x7 Matrix (set 40)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into

a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NumPy Math, Broadcasting & Linear Algebra, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example. first anchor Project 41 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

🧭 Skill Roadmap

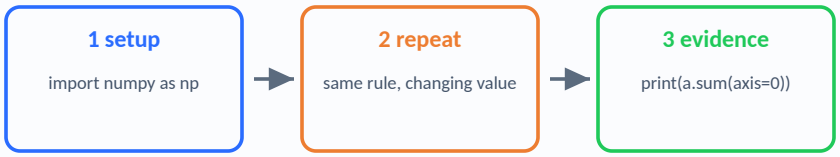
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 41 changes the surface problem while preserving the underlying method. progress check

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: vectorised math, broadcasting rules, matrix operations.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vectorised math, broadcasting rules, matrix operations.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NumPy Math, Broadcasting & Linear Algebra becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 40): connect the surface task to the chapter skill, vectorised math, broadcasting rules, matrix operations.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix 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 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Forgetting to apply the *same* fitted transform to the test set.
- Shuffling time-series data before splitting — future leaks into past.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 41 Summarise arange 1..12

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..12 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Math, Broadcasting & Linear Algebra, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 40, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 42 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 40: column sums of a 7x7 matrix. Project 42 changes the surface problem while preserving the underlying method.

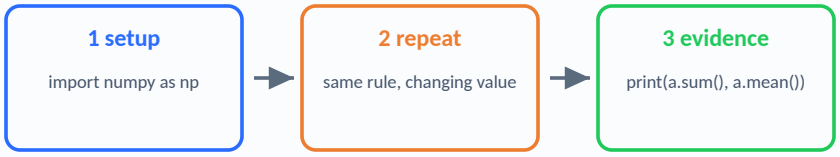
progress check

Reading target: summarise arange 1..12 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: vectorised math, broadcasting rules, matrix operations.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vectorised math, broadcasting rules, matrix operations.
2. Apply the rule	`a = np.arange(1, 13)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Math, Broadcasting & Linear Algebra becomes practical when you can build the syntax 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 = np.arange(1, 13)`

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. 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	Summarise arange 1..12: connect the surface task to the chapter skill, vectorised math, broadcasting rules, matrix operations.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 13) | Evidence: print(a.sum(), a.mean())

Reader to Writer Bridge

1. Read: explain summarise arange 1..12 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..12 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 13)
print(a.sum(), a.mean())
```

```
output
78 6.5
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- One-hot encoding after splitting inconsistently, so train/test columns differ.
- Dropping rows with any NaN and silently losing most of the data.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 42 Sort an Array (set 42)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NumPy Math, Broadcasting & Linear Algebra, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 41, where you practised summarise arange 1..12, and asks you to reuse the same reasoning pattern. reuse the pattern Project 43 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 41: summarise arange 1..12. Project 43 changes the surface problem while preserving the underlying method. progress check

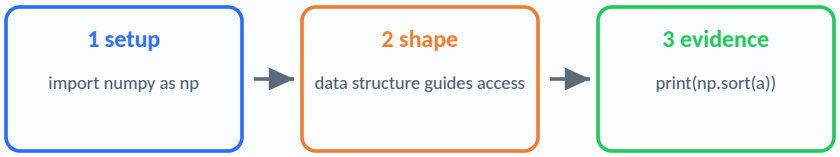
Reading target: sort an array is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.

- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: vectorised math, broadcasting rules, matrix operations.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vectorised math, broadcasting rules, matrix operations.
2. Apply the rule	`a = np.array([18, 29, 40, 51, 62])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Math, Broadcasting & Linear Algebra becomes practical when you can build the syntax 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 = np.array([18, 29, 40, 51, 62])`

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. 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	Sort an Array (set 42): connect the surface task to the chapter skill, vectorised math, broadcasting rules, matrix operations.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([18, 29, 40, 51, 62]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([18, 29, 40, 51, 62])
print(np.sort(a))
```

```
output
[18 29 40 51 62]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Imputing with the mean in the presence of extreme outliers.
- Scaling the target variable when only features needed it.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 43 Scale arange 1..6 by 8

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..6 by 8 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Math, Broadcasting & Linear Algebra, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 42, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 44 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

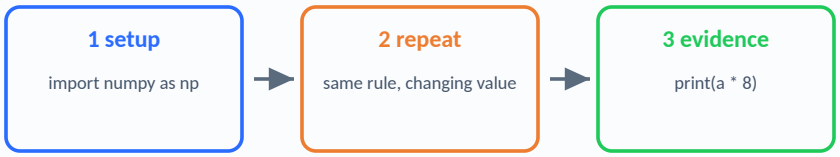
Skill step: pushes the same idea into a less automatic situation. It builds on Project 42: sort an array. Project 44 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..6 by 8 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: vectorised math, broadcasting rules, matrix operations.
- Evidence: `print(a * 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vectorised math, broadcasting rules, matrix operations.
2. Apply the rule	`a = np.arange(1, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 8)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Math, Broadcasting & Linear Algebra becomes practical when you can build the syntax 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 = np.arange(1, 7)`

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. 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	Scale arange 1..6 by 8: connect the surface task to the chapter skill, vectorised math, broadcasting rules, matrix operations.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 7) | Evidence: print(a * 8)

Reader to Writer Bridge

1. Read: explain scale arange 1..6 by 8 in one sentence before touching the 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(a * 8)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..6 by 8 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 7)
print(a * 8)
```

```
output
[ 8 16 24 32 40 48]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 8)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Fitting the scaler on the full dataset before splitting — leakage.
- Forgetting to apply the *same* fitted transform to the test set.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 44 Boolean Mask: Above Mean (set 44)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NumPy Math, Broadcasting & Linear Algebra, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 43, where you practised scale arange 1..6 by 8, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

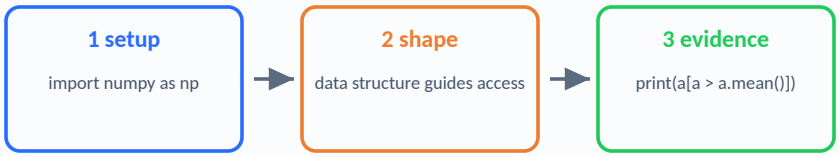
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 43: scale arange 1..6 by 8. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: vectorised math, broadcasting rules, matrix operations.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vectorised math, broadcasting rules, matrix operations.
2. Apply the rule	`a = np.array([29, 34, 39, 44, 49, 4])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NumPy Math, Broadcasting & Linear Algebra becomes practical when you can build the syntax 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 = np.array([29, 34, 39, 44, 49, 4])`

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. 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	Boolean Mask: Above Mean (set 44): connect the surface task to the chapter skill, vectorised math, broadcasting rules, matrix operations.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([29, 34, 39, 44, 49, 4]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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, 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```
import numpy as np

a = np.array([29, 34, 39, 44, 49, 4])
print(a[a > a.mean()])
```

```
output
[34 39 44 49]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Standardising one-hot columns and destroying their 0/1 meaning.
- One-hot encoding after splitting inconsistently, so train/test columns differ.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 5 mini-projects, NumPy Math, Broadcasting & Linear Algebra should feel like a reusable Python move: vectorised math, broadcasting rules, matrix operations. *chapter checkpoint*

1 Read	2 Write	3 Explain
Open any project from Projects 40-44 and mark setup, rule, and evidence without looking back at the explanation.	Create one sibling program that keeps the same logic but changes the purpose, data, or output wording. *transfer proves learning*	Say what this chapter builds on and where it leads: Next, Chapter 3 (pandas Essentials) 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

Recall · Meaning · Structure · Transfer

2.1.Q1 Transfer — Bug hunt: a pipeline scales the whole dataset, then splits, and test accuracy looks great — until production. What leaked?

2.1.Q2 Meaning — Bug hunt: after one-hot encoding train and test separately, the model crashes on test. Why?

2.1.Q3 Structure — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

2.1.Q4 Transfer — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

2.1.Q5 Recall — What is the most important edge case in NumPy Math, Broadcasting & Linear Algebra?

2.1.Q6 Meaning What proves the skill is transferable?

Answer Key – Chapter 2 – NumPy Math, Broadcasting & Linear Algebra

2.1.Q1. The scaler’ s mean/std were computed with test rows included — test information reached training; fit on train only.

2.1.Q2. Test had unseen/missing categories, so the column sets differ — fit the encoder on train and transform test with it.

2.1.Q3. A — Broadcasting and linear algebra turn many scalar operations into one readable vector expression.

2.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

2.1.Q5. Broadcasting bugs are quiet when a shape is compatible but semantically wrong.

2.1.Q6. Change bias to a column vector and explain the different result.

Chapter 3 · pandas Essentials

Series, DataFrames, selection, groupby — tabular data fluency.

Chapter Skill Map

What this chapter adds: In pandas Essentials, the focus is series, dataframes, selection, groupby — tabular data fluency. You practise it across 9 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 3 assumes the reading habits from Chapter 2: NumPy Math, Broadcasting & Linear Algebra. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 4, Data Preparation & Feature Engineering, reuses this chapter's discipline with a new kind of problem.

1 **Projects 45-47**
Recognise the pattern and read the code slowly. *name it first*

2 **Projects 48-50**
Trace similar programs with less prompting and explain what changed.

3 **Projects 51-53**
Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

pandas Essentials: what problem does this solve for a working programmer?

name the move before typing

pandas makes tabular data inspectable: select columns, filter rows, group records and compute features.

This chapter's focus is: Series, DataFrames, selection, groupby — tabular data fluency. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
import pandas as pd

df = pd.DataFrame({"team": ["A", "A", "B"], "score": [10, 14, 7]})
print(df.groupby("team")["score"].mean().to_dict())
```

```
output
{'A': 12.0, 'B': 7.0}
```

groupby builds one mean per team, then to_dict exposes the result compactly. *predict first, verify second*

DATAFRAME

A labelled table with rows and columns.

SERIES

One labelled column or vector in pandas.

GROUPBY

Split rows by key, apply an operation, combine results.

INDEX

Labels pandas uses to align rows and columns.

Beginner model

At beginner level, pandas Essentials should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Production feature work is usually table work before it becomes model work.

The edge case to respect

Index alignment can surprise beginners: pandas matches labels, not just positions.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Add a second numeric column and compute grouped means for both columns.

Example 1

Trace the opening pandas Essentials example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read pandas Essentials as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Add a second numeric column and compute grouped means for both columns.

⚠ Common Mistakes

- Copying the surface syntax for pandas Essentials without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Index alignment can surprise beginners: pandas matches labels, not just positions.
- Trusting the output before predicting it by hand.

This chapter contains 9 hands-on mini-projects (Projects 45–53). Type each one out, run it, and experiment.

Project 45 Group-By Aggregation

The Idea Behind It

pandas is where most data work begins: load, clean, filter, group and feature-engineer tabular data before modelling. The deeper point is learning how Python turns a small intention like group-by aggregation into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 46 will change the surface task again, so treat this one as a checkpoint: predict the

output before you look ahead. *predict first*

Skill Roadmap

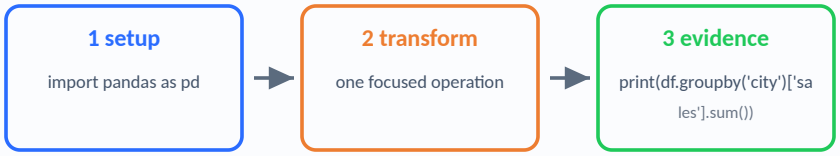
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 46 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import pandas as pd` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is tabular: labels choose the data, an operation transforms a column/group, and the final expression reports the answer. *table question -> table answer*
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(df.groupby('city')['sales'].sum())` exposes the result; match it 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 pandas as pd`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`df = pd.DataFrame({'city': ['NY','LA','NY','LA'],`	This line asks the table a question. The labels, grouping, or column operation decide what answer can be reported. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(df.groupby('city')['sales'].sum())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `df = pd.DataFrame({'city': ['NY','LA','NY','LA'],`

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	Group-By Aggregation: connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: pandas, pd, df, DataFrame, city. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import pandas as pd | Core: df = pd.DataFrame({'city': ['NY','LA','NY','LA'], | Evidence: print(df.groupby('city')['sales'].sum())

Reader to Writer Bridge

1. Read: explain group-by aggregation 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 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(df.groupby('city')['sales'].sum())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import pandas as pd` and say why 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-by aggregation and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

groupby buckets rows by a key, then aggregates each bucket.

```
import pandas as pd

df = pd.DataFrame({'city': ['NY','LA','NY','LA'],
                   'sales': [10, 7, 5, 8]})
print(df.groupby('city')['sales'].sum())
```

```
output
city
LA    15
NY    15
Name: sales, dtype: int64
```

How it works:

1. Start at `import pandas as pd`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(df.groupby('city')['sales'].sum())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Sort the result.

⚠ Common Mistakes

- Forgetting groupby returns a grouped object until you aggregate.
- Averaging metrics across folds without their spread.
- Optimising precision when the costly error is a miss (needs recall).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 46 Create a Derived Column

The Idea Behind It

pandas is where most data work begins: load, clean, filter, group and feature-engineer tabular data before modelling. The deeper point is learning how Python turns a small intention like create a derived column into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 45, where you practised group-by aggregation, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 47 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 45: group-by aggregation. Project 47 changes the surface problem while preserving the underlying method.

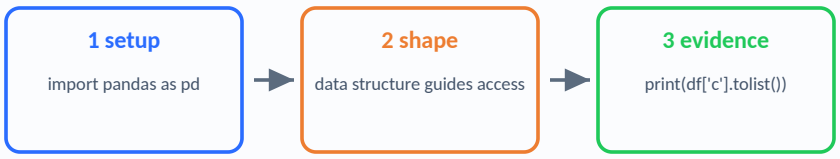
progress check

Reading target: create a derived column is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import pandas as pd` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is tabular: labels choose the data, an operation transforms a column/group, and the final expression reports the answer. table question -> table answer
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(df['c'].tolist())` exposes the result; match it 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 pandas as pd`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})`	This line asks the table a question. The labels, grouping, or column operation decide what answer can be reported. slow line
3. Check the evidence	`print(df['c'].tolist())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})`

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

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	Create a Derived Column: connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: pandas, pd, df, DataFrame, tolist. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import pandas as pd | Core: df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]}) | Evidence: print(df['c'].tolist())

🛠️ Reader to Writer Bridge

1. Read: explain create a derived column in one sentence before touching the 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(df['c'].tolist())`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import pandas as pd` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of create a derived column and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Column arithmetic is vectorised, like NumPy.

```
import pandas as pd

df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})
df['c'] = df['a'] + df['b']
print(df['c'].tolist())
```

```
output
[5, 7, 9]
```

How it works:

1. Start at `import pandas as pd`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(df['c'].tolist())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add a second aggregation.

⚠ Common Mistakes

- Chained indexing warnings — prefer .loc.
- Reporting test metrics after tuning on the test set.
- Quoting accuracy on a 99:1 imbalanced dataset.

Skills gained choosing and reading data structures predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 47 Boolean Filtering of Rows

The Idea Behind It

pandas is where most data work begins: load, clean, filter, group and feature-engineer tabular data before modelling. The deeper point is learning how Python turns a small intention like boolean filtering of rows into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 46, where you practised create a derived column, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 48 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 46: create a derived column. Project 48 changes the surface problem while preserving the underlying method.

progress check

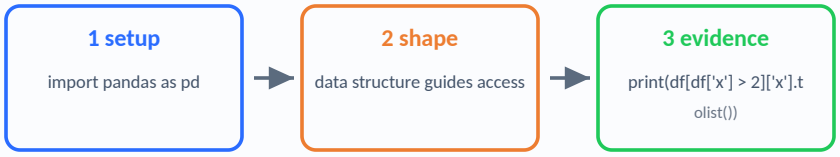
Reading target: boolean filtering of rows is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

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

- Core logic: The logic is tabular: labels choose the data, an operation transforms a column/group, and the final expression reports the answer. table question -> table answer
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(df[df['x'] > 2]['x'].tolist())` exposes the result; match it 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 pandas as pd`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`df = pd.DataFrame({'x': [1, 2, 3, 4, 5]})`	This line asks the table a question. The labels, grouping, or column operation decide what answer can be reported. slow line
3. Check the evidence	`print(df[df['x'] > 2]['x'].tolist())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `df = pd.DataFrame({'x': [1, 2, 3, 4, 5]})`

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

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	Boolean Filtering of Rows: connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: pandas, pd, df, DataFrame, tolist. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import pandas as pd | Core: df = pd.DataFrame({'x': [1, 2, 3, 4, 5]}) | Evidence: print(df[df['x'] > 2]['x'].tolist())

🛠️ Reader to Writer Bridge

1. Read: explain boolean filtering of rows in one sentence before touching the 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(df[df['x'] > 2]['x'].tolist())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import pandas as pd` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of boolean filtering of rows and name what must stay the 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 boolean Series selects the rows that pass the test.

```
import pandas as pd

df = pd.DataFrame({'x': [1, 2, 3, 4, 5]})
print(df[df['x'] > 2]['x'].tolist())
```

```
output
[3, 4, 5]
```

How it works:

1. Start at `import pandas as pd`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(df[df['x'] > 2]['x'].tolist())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Sort the result.

⚠ Common Mistakes

- Forgetting groupby returns a grouped object until you aggregate.
- Reading a confusion matrix with the axes transposed.
- Using R^2 to compare models on different subsets of data.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 48 Mean per Group

The Idea Behind It

pandas is where most data work begins: load, clean, filter, group and feature-engineer tabular data before modelling. The deeper point is learning how Python turns a small intention like mean per group into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 47, where you practised boolean filtering of rows, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 49 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 47: boolean filtering of rows. Project 49 changes the surface problem while preserving the underlying method. *progress check*

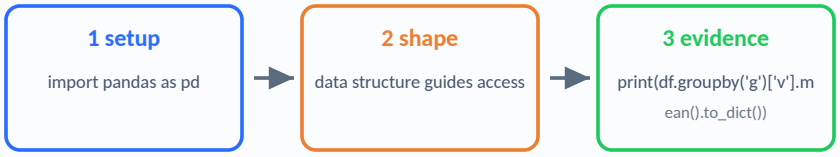
Reading target: mean per group is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import pandas as pd` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The logic is tabular: labels choose the data, an operation transforms a column/group, and the final expression reports the answer. *table question -> table answer*
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.

- Evidence: `print(df.groupby('g')['v'].mean().to_dict())` exposes the result; match it 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 pandas as pd`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`df = pd.DataFrame({'g': ['a','a','b'], 'v': [10, 20, 30]})`	This line asks the table a question. The labels, grouping, or column operation decide what answer can be reported. *slow line*
3. Check the evidence	`print(df.groupby('g')['v'].mean().to_dict())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `df = pd.DataFrame({'g': ['a','a','b'], 'v': [10, 20, 30]})`

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

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	Mean per Group: connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: pandas, pd, df, DataFrame, groupby. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import pandas as pd | Core: df = pd.DataFrame({'g': ['a','a','b'], 'v': [10, 20, 30]}) | Evidence: print(df.groupby('g')['v'].mean().to_dict())

🛠️ Reader to Writer Bridge

1. Read: explain mean per group in one sentence before touching the 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(df.groupby('g')['v'].mean().to_dict())`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `import pandas as pd` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of mean per group and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Aggregations return a Series indexed by the group key.

```
import pandas as pd
```

```
df = pd.DataFrame({'g': ['a','a','b'], 'v': [10, 20, 30]})
print(df.groupby('g')['v'].mean().to_dict())
```

```
output
{'a': 15.0, 'b': 30.0}
```

How it works:

1. Start at `import pandas as pd`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(df.groupby('g')['v'].mean().to_dict())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add a second aggregation.

⚠ Common Mistakes

- Chained indexing warnings — prefer .loc.
- Computing F1 as the arithmetic (not harmonic) mean.
- Averaging metrics across folds without their spread.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 49 Column Sums of a 7x7 Matrix (set 45)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 48, where you practised mean per group, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 50 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

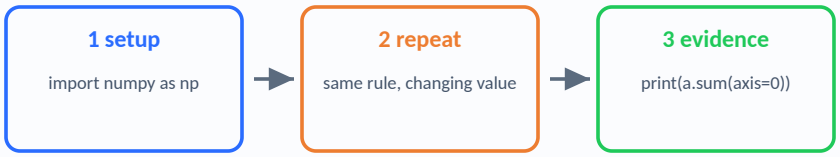
Skill step: moves you from recognition toward independent tracing. It builds on Project 48: mean per group. Project 50 changes the surface problem while preserving the underlying method. *progress check*

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: pandas Essentials becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 45): connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools; loops repeat a rule.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Comparing MAE and RMSE across different target scales.
- Reporting test metrics after tuning on the test set.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 50 Summarise arange 1..13

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..13 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 49, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 51 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 49: column sums of a 7x7 matrix. Project 51 changes the surface problem while preserving the underlying method.

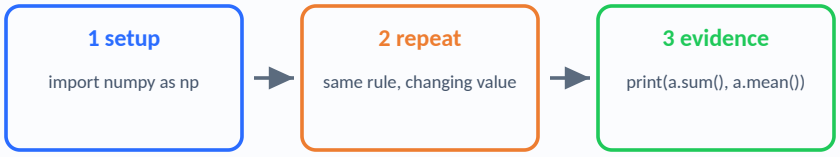
progress check

Reading target: summarise arange 1..13 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`a = np.arange(1, 14)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: pandas Essentials becomes practical when you can build the syntax 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 = np.arange(1, 14)`

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. 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	Summarise arange 1..13: connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 14) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..13 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..13 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 14)
print(a.sum(), a.mean())
```

```
output
91 7.0
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Optimising precision when the costly error is a miss (needs recall).
- Reading a confusion matrix with the axes transposed.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 51 Sort an Array (set 47)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project pushes the same idea into a less automatic situation.
- It builds on Project 50, where you practised summarise arange 1..13, and asks you to reuse the same reasoning pattern. reuse the pattern Project 52 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 50: summarise arange 1..13. Project 52 changes the surface problem while preserving the underlying method. progress check

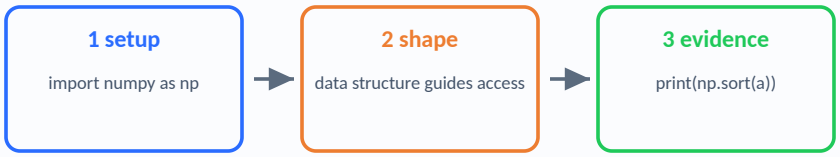
Reading target: sort an array is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.

- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`a = np.array([53, 64, 75, 86, 7])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: pandas Essentials becomes practical when you can build the syntax 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 = np.array([53, 64, 75, 86, 7])`

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. 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	Sort an Array (set 47): connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import numpy as np | Core: a = np.array([53, 64, 75, 86, 7]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([53, 64, 75, 86, 7])
print(np.sort(a))
```

```
output
[ 7 53 64 75 86]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Quoting accuracy on a 99:1 imbalanced dataset.
- Computing F1 as the arithmetic (not harmonic) mean.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 52 Scale arange 1..7 by 4

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..7 by 4 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 51, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 53 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

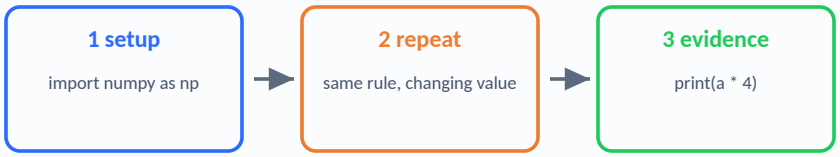
Skill step: pushes the same idea into a less automatic situation. It builds on Project 51: sort an array. Project 53 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..7 by 4 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(a * 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`a = np.arange(1, 8)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 4)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: pandas Essentials becomes practical when you can build the syntax 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 = np.arange(1, 8)`

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. 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	Scale arange 1..7 by 4: connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 8) | Evidence: print(a * 4)

Reader to Writer Bridge

1. Read: explain scale arange 1..7 by 4 in one sentence before touching the 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(a * 4)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..7 by 4 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 8)
print(a * 4)
```

```
output
[ 4  8 12 16 20 24 28]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 4)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Using R^2 to compare models on different subsets of data.
- Comparing MAE and RMSE across different target scales.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 53 Boolean Mask: Above Mean (set 49)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, pandas Essentials, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 52, where you practised scale arange 1..7 by 4, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

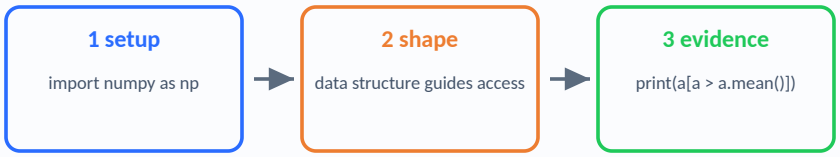
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 52: scale arange 1..7 by 4. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: series, dataframes, selection, groupby — tabular data fluency.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: series, dataframes, selection, groupby — tabular data fluency.
2. Apply the rule	`a = np.array([44, 49, 4, 9, 14, 19])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: pandas Essentials becomes practical when you can build the syntax 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 = np.array([44, 49, 4, 9, 14, 19])`

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. 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	Boolean Mask: Above Mean (set 49): connect the surface task to the chapter skill, series, dataframes, selection, groupby — tabular data fluency.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Tabular data: rows, columns, labels, groups, and derived summaries.
Evidence	Entry: import numpy as np | Core: a = np.array([44, 49, 4, 9, 14, 19]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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, 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```
import numpy as np

a = np.array([44, 49, 4, 9, 14, 19])
print(a[a > a.mean()])
```

```
output
[44 49]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Averaging metrics across folds without their spread.
- Optimising precision when the costly error is a miss (needs recall).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 9 mini-projects, pandas Essentials should feel like a reusable Python move: series, dataframes, selection, groupby — tabular data fluency. *chapter checkpoint*

1 Read	2 Write	3 Explain
Open any project from Projects 45-53 and mark setup, rule, and evidence without looking back at the explanation.	Create one sibling program that keeps the same logic but changes the purpose, data, or output wording. *transfer proves learning*	Say what this chapter builds on and where it leads: Next, Chapter 4 (Data Preparation & Feature Engineering) 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

Recall · Meaning · Structure · Transfer

3.1.Q1 Meaning — Bug hunt: F1 computed as (precision + recall) / 2 gives 0.5 for precision 1.0, recall 0.0. What does the true F1 say?

3.1.Q2 Meaning — Bug hunt: a "97% accurate" tumour screen misses 60% of tumours. Which two numbers expose it?

3.1.Q3 Structure — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

3.1.Q4 Transfer — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

3.1.Q5 Recall — What is the most important edge case in pandas Essentials?

3.1.Q6 Meaning What proves the skill is transferable?

Answer Key – Chapter 3 – pandas Essentials

3.1.Q1. 0 — the harmonic mean collapses when either component is 0; that is exactly why F1 uses it.

3.1.Q2. Recall (0.4) and the false-negative count from the confusion matrix — accuracy hides them under class imbalance.

3.1.Q3. A — pandas makes tabular data inspectable: select columns, filter rows, group records and compute features.

3.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

3.1.Q5. Index alignment can surprise beginners: pandas matches labels, not just positions.

3.1.Q6. Add a second numeric column and compute grouped means for both columns.

Chapter 4 · Data Preparation & Feature Engineering

Cleaning, scaling, encoding, splitting — where projects are won.

Chapter Skill Map

What this chapter adds: In Data Preparation & Feature Engineering, the focus is cleaning, scaling, encoding, splitting — where projects are won. You practise it across 11 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 4 assumes the reading habits from Chapter 3: pandas Essentials. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 5, Evaluation Metrics from Scratch, reuses this chapter's discipline with a new kind of problem.

1 **Projects 54-56**
Recognise the pattern and read the code slowly. *name it first*

2 **Projects 57-59**
Trace similar programs with less prompting and explain what changed.

3 **Projects 60-64**
Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Data Preparation & Feature Engineering: what problem does this solve for a working programmer?

name the move before typing

Data preparation is where projects are won: clean, scale, encode and split before training.

This chapter's focus is: Cleaning, scaling, encoding, splitting — where projects are won. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```python
import numpy as np

x = np.array([10.0, 20.0, 30.0])
z = (x - x.mean()) / x.std()
print(np.round(z, 2))
```

```
output
[-1.22  0.    1.22]
```

Each value is measured by how many standard deviations it sits from the mean. *predict first, verify second*

FEATURE

A model input column or derived signal.

SCALING

Transforming numeric ranges so one feature does not dominate by units alone.

ENCODING

Turning categories or text into numeric representations.

DATA LEAKAGE

Letting validation/test information influence training preparation.

Beginner model

At beginner level, Data Preparation & Feature Engineering should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

A simple model with clean features usually beats a fancy model with sloppy data.

The edge case to respect

Fit scalers on training data only; otherwise validation data leaks into the model.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Split a tiny table into train/test first, then compute the mean only on train.

Example 1

Trace the opening Data Preparation & Feature Engineering example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read Data Preparation & Feature Engineering as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Split a tiny table into train/test first, then compute the mean only on train.

⚠ Common Mistakes

- Copying the surface syntax for Data Preparation & Feature Engineering without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Fit scalers on training data only; otherwise validation data leaks into the model.
- Trusting the output before predicting it by hand.

This chapter contains 11 hands-on mini-projects (Projects 54–64). Type each one out, run it, and experiment.

Project 54 Min-Max Normalisation

💡 The Idea Behind It

Models are only as good as their data. Preparation — scaling, imputing, encoding, splitting — is most of the job. The deeper point is learning how Python turns a small intention like min-max normalisation into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.

- It starts this chapter's project sequence by turning the chapter focus into a small runnable example. *first anchor* Project 55 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

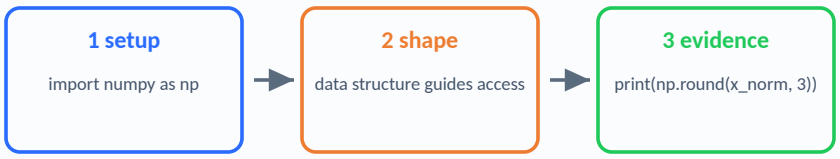
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 55 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(np.round(x_norm, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`x = np.array([10, 20, 30], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(np.round(x_norm, 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.array([10, 20, 30], dtype=float)`

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. 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	Min-Max Normalisation: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, array, dtype, x_norm. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: x = np.array([10, 20, 30], dtype=float) | Evidence: print(np.round(x_norm, 3))

Reader to Writer Bridge

1. Read: explain min-max normalisation 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 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(np.round(x_norm, 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of min-max normalisation and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Rescale features to the 0–1 range so no feature dominates.

```python
import numpy as np

x = np.array([10, 20, 30], dtype=float)
x_norm = (x - x.min()) / (x.max() - x.min())
print(np.round(x_norm, 3))
```

```
output
[0.  0.5 1. ]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(x_norm, 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a 70/30 split.

⚠ Common Mistakes

- Fitting the scaler on test data (leakage).

- argmax on the flattened array when you meant a row-wise argmax.
- Trusting a remembered shape instead of predicting then checking.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 55 Impute Missing Values

The Idea Behind It

Models are only as good as their data. Preparation — scaling, imputing, encoding, splitting — is most of the job. The deeper point is learning how Python turns a small intention like impute missing values into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 54, where you practised min-max normalisation, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 56 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 54: min-max normalisation. Project 56 changes the surface problem while preserving the underlying method.

progress check

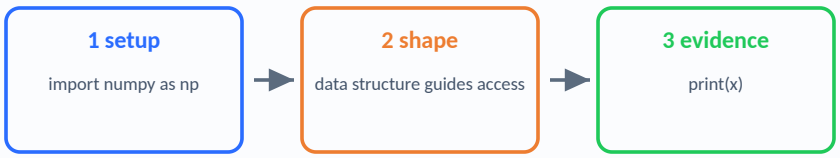
Reading target: impute missing values is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(x)` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`x = np.array([1.0, np.nan, 3.0, np.nan, 5.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(x)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.array([1.0, np.nan, 3.0, np.nan, 5.0])`

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. 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	Impute Missing Values: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, array, nan, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: x = np.array([1.0, np.nan, 3.0, np.nan, 5.0]) | Evidence: print(x)

🛠️ Reader to Writer Bridge

1. Read: explain impute missing 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 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(x)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of impute missing 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)

Replace NaNs with the column mean — a basic imputation strategy.

```
import numpy as np

x = np.array([1.0, np.nan, 3.0, np.nan, 5.0])
mean = np.nanmean(x)
x = np.where(np.isnan(x), mean, x)
print(x)
```

```
output
[1. 3. 3. 3. 5.]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(x)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Use z-score scaling instead of min-max.

⚠ Common Mistakes

- Forgetting to set a seed, so results aren't reproducible.
- Chained fancy indexing creating copies you then mutate in vain.
- np.dot on shapes that "work" but mean the wrong contraction.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 56 Label Encoding

💡 The Idea Behind It

Models are only as good as their data. Preparation — scaling, imputing, encoding, splitting — is most of the job. The deeper point is learning how Python turns a small intention like label encoding into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 55, where you practised impute missing values, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 57 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 55: impute missing values. Project 57 changes the surface problem while preserving the underlying method.

progress check

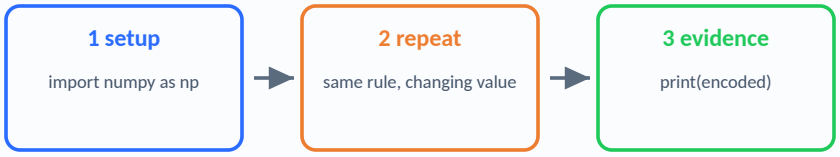
Reading target: label encoding is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to compress a readable loop into a collection-building expression. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.

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

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`encoded = [classes.index(l) for l in labels]`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(encoded)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `encoded = [classes.index(l) for l in labels]`

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. 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	Label Encoding: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, labels, cat, dog. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: encoded = [classes.index(l) for l in labels] | Evidence: print(encoded)

🛠️ Reader to Writer Bridge

1. Read: explain label encoding in one sentence before touching the 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(encoded)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of label encoding and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Map categorical strings to integer ids for the model.

```python
import numpy as np

labels = ['cat', 'dog', 'cat', 'bird']
classes = sorted(set(labels))
encoded = [classes.index(l) for l in labels]
print(classes)
print(encoded)
```

```
output
```

```
['bird', 'cat', 'dog']
[1, 2, 1, 0]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(encoded)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a 70/30 split.

⚠ Common Mistakes

- Fitting the scaler on test data (leakage).
- Masking with integers (a[mask] where mask is 0/1 ints indexes positions!).
- Forgetting keepdims and breaking later broadcasting.

Skills gained tracing repeated work | choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 57 Train/Test Split

The Idea Behind It

Models are only as good as their data. Preparation — scaling, imputing, encoding, splitting — is most of the job. The deeper point is learning how Python turns a small intention like train/test split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 56, where you practised label encoding, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 58 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

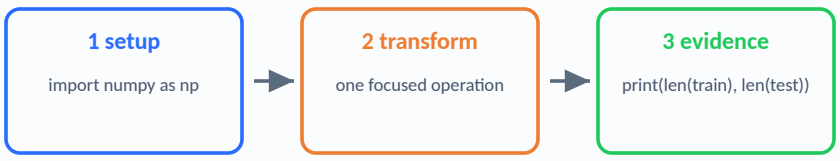
Skill step: moves you from recognition toward independent tracing. It builds on Project 56: label encoding. Project 58 changes the surface problem while preserving the underlying method. *progress check*

Reading target: train/test split is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(len(train), len(test))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`rng = np.random.default_rng(42)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(len(train), len(test))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `rng = np.random.default_rng(42)`

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	Train/Test Split: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, rng, random, default_rng. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: rng = np.random.default_rng(42) | Evidence: print(len(train), len(test))

Reader to Writer Bridge

1. Read: explain train/test split in one sentence before touching the 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(len(train), len(test))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of train/test split and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Shuffle indices, then slice 80/20 into train and test sets.

```
import numpy as np

rng = np.random.default_rng(42)
idx = rng.permutation(10)
split = int(0.8 * 10)
train, test = idx[:split], idx[split:]
print(len(train), len(test))
```

```
output
8 2
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(len(train), len(test))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.

3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Use z-score scaling instead of min-max.

⚠ Common Mistakes

- Forgetting to set a seed, so results aren't reproducible.
- Using == on floats after arithmetic instead of np.isclose.
- argmax on the flattened array when you meant a row-wise argmax.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 58 Min-Max Normalisation

The Idea Behind It

Models are only as good as their data. Preparation — scaling, imputing, encoding, splitting — is most of the job. The deeper point is learning how Python turns a small intention like min-max normalisation into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 57, where you practised train/test split, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 59 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

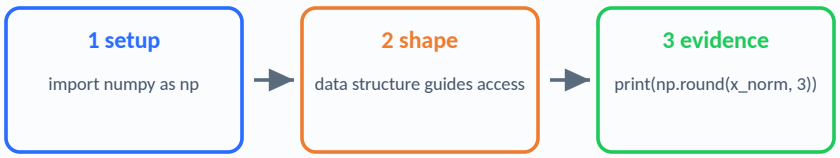
Skill step: moves you from recognition toward independent tracing. It builds on Project 57: train/test split. Project 59 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(np.round(x_norm, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`x = np.array([10, 20, 30, 40], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(x_norm, 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.array([10, 20, 30, 40], dtype=float)`

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. 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	Min-Max Normalisation: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, array, dtype, x_norm. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: x = np.array([10, 20, 30, 40], dtype=float) | Evidence: print(np.round(x_norm, 3))

🛠️ Reader to Writer Bridge

1. Read: explain min-max normalisation in one sentence before touching the 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(np.round(x_norm, 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

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

From Concept to Code (Python)

Rescale features to the 0–1 range so no feature dominates.

```
import numpy as np

x = np.array([10, 20, 30, 40], dtype=float)
x_norm = (x - x.min()) / (x.max() - x.min())
print(np.round(x_norm, 3))
```

```
output
[0.    0.333 0.667 1.   ]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(x_norm, 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a 70/30 split.

⚠ Common Mistakes

- Fitting the scaler on test data (leakage).
- Broadcasting (3,) against (3, 1) and getting (3, 3) by surprise.
- Chained fancy indexing creating copies you then mutate in vain.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 59 Min-Max Normalisation

The Idea Behind It

Models are only as good as their data. Preparation — scaling, imputing, encoding, splitting — is most of the job. The deeper point is learning how Python turns a small intention like min-max normalisation into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 58, where you practised min-max normalisation, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 60 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 58: min-max normalisation. Project 60 changes the surface problem while preserving the underlying method.

progress check

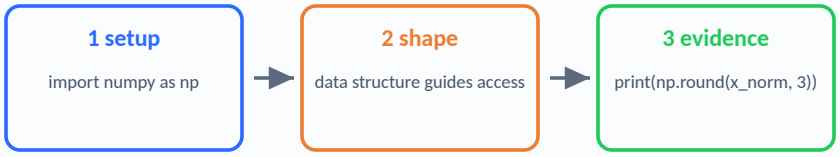
Reading target: min-max normalisation is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.

- Evidence: `print(np.round(x_norm, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`x = np.array([10, 20, 30, 40, 50], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(x_norm, 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.array([10, 20, 30, 40, 50], dtype=float)`

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. 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	Min-Max Normalisation: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, array, dtype, x_norm. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: x = np.array([10, 20, 30, 40, 50], dtype=float) | Evidence: print(np.round(x_norm, 3))

🛠️ Reader to Writer Bridge

1. Read: explain min-max normalisation in one sentence before touching the 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(np.round(x_norm, 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of min-max normalisation and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Rescale features to the 0–1 range so no feature dominates.

```
import numpy as np

x = np.array([10, 20, 30, 40, 50], dtype=float)
x_norm = (x - x.min()) / (x.max() - x.min())
print(np.round(x_norm, 3))
```

```
output
```

```
[0.   0.25 0.5  0.75 1.  ]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(x_norm, 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Use z-score scaling instead of min-max.

⚠ Common Mistakes

- Forgetting to set a seed, so results aren't reproducible.
- Trusting a remembered shape instead of predicting then checking.
- Masking with integers (a[mask] where mask is 0/1 ints indexes positions!).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 60 Column Sums of a 7x7 Matrix (set 50)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 59, where you practised min-max normalisation, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 61 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 59: min-max normalisation. Project 61 changes the surface problem while preserving the underlying method.

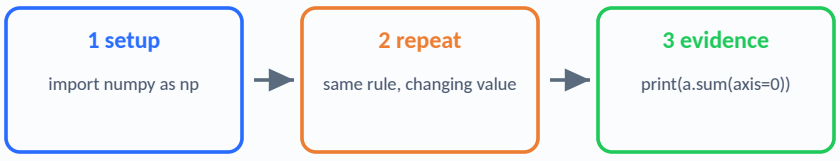
progress check

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 50): connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- np.dot on shapes that "work" but mean the wrong contraction.
- Using == on floats after arithmetic instead of np.isclose.

Skills gained: tracing repeated work | choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 61 Summarise arange 1..14

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..14 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 60, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 62 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

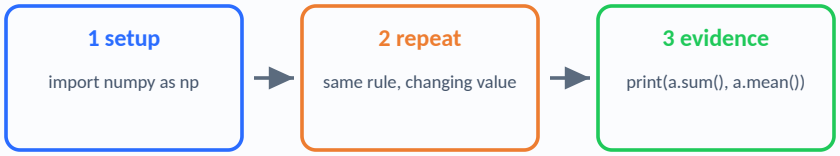
Skill step: pushes the same idea into a less automatic situation. It builds on Project 60: column sums of a 7x7 matrix. Project 62 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..14 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`a = np.arange(1, 15)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.arange(1, 15)`

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. 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	Summarise arange 1..14: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 15) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..14 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..14 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```
import numpy as np

a = np.arange(1, 15)
print(a.sum(), a.mean())
```

```
output
105 7.5
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Forgetting keepdims and breaking later broadcasting.
- Broadcasting (3,) against (3, 1) and getting (3, 3) by surprise.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 62 Sort an Array (set 52)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 61, where you practised summarise arange 1..14, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 63 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

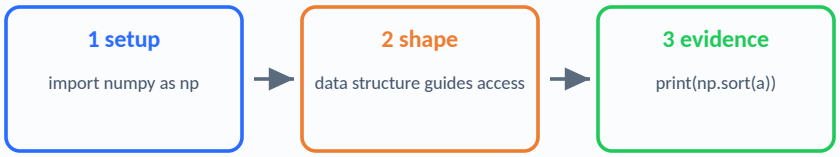
Skill step: pushes the same idea into a less automatic situation. It builds on Project 61: summarise arange 1..14. Project 63 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`a = np.array([88, 9, 20, 31, 42])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.array([88, 9, 20, 31, 42])`

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. 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	Sort an Array (set 52): connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([88, 9, 20, 31, 42]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([88, 9, 20, 31, 42])
print(np.sort(a))
```

```
output
[ 9 20 31 42 88]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- argmax on the flattened array when you meant a row-wise argmax.
- Trusting a remembered shape instead of predicting then checking.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 63 Scale arange 1..4 by 9

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..4 by 9 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.

- It builds on Project 62, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 64 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

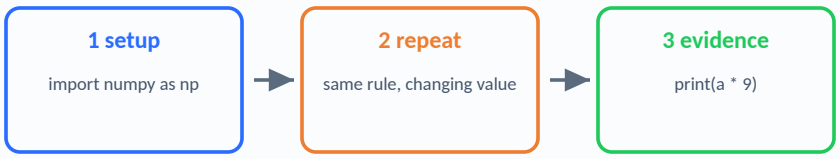
Skill step: pushes the same idea into a less automatic situation. It builds on Project 62: sort an array. Project 64 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..4 by 9 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(a * 9)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`a = np.arange(1, 5)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a * 9)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.arange(1, 5)`

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. 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	Scale arange 1..4 by 9: connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 5) | Evidence: print(a * 9)

Reader to Writer Bridge

1. Read: explain scale arange 1..4 by 9 in one sentence before touching the 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(a * 9)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..4 by 9 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 5)
print(a * 9)
```

```
output
[ 9 18 27 36]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 9)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Chained fancy indexing creating copies you then mutate in vain.
- np.dot on shapes that "work" but mean the wrong contraction.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 64 Boolean Mask: Above Mean (set 54)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Data Preparation & Feature Engineering, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 63, where you practised scale arange 1..4 by 9, and asks you to reuse the same reasoning pattern. reuse the pattern After this, you should be able to explain the chapter pattern without leaning on the exact values used here. own the pattern

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 63: scale arange 1..4 by 9. By the end of this chapter, you should be able to explain the pattern without copying the exact example. progress check

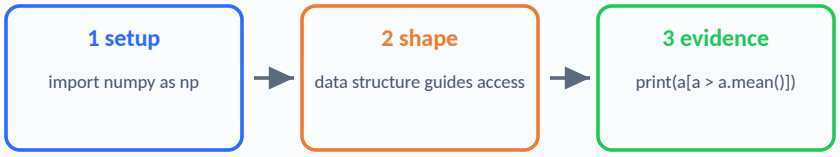
Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

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

- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: cleaning, scaling, encoding, splitting — where projects are won.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: cleaning, scaling, encoding, splitting — where projects are won.
2. Apply the rule	`a = np.array([9, 14, 19, 24, 29, 34])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Data Preparation & Feature Engineering becomes practical when you can build the syntax 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 = np.array([9, 14, 19, 24, 29, 34])`

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. 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	Boolean Mask: Above Mean (set 54): connect the surface task to the chapter skill, cleaning, scaling, encoding, splitting — where projects are won.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([9, 14, 19, 24, 29, 34]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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, 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```
import numpy as np
```

```
a = np.array([9, 14, 19, 24, 29, 34])
print(a[a > a.mean()])
```

```
output
[24 29 34]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Masking with integers (a[mask] where mask is 0/1 ints indexes positions!).
- Forgetting keepdims and breaking later broadcasting.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 11 mini-projects, Data Preparation & Feature Engineering should feel like a reusable Python move: cleaning, scaling, encoding, splitting — where projects are won. *chapter checkpoint*

1 Read
Open any project from Projects 54-64 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 (Evaluation Metrics from Scratch) 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

Recall · Meaning · Structure · Transfer

4.1.Q1 Recall — Bug hunt: `a[np.array([0, 1, 0, 1])]` on a length-4 array did not filter — it selected rows 0,1,0,1. What was intended?

4.1.Q2 Meaning — Bug hunt: `v + w` for v of (3,) and w of (3, 1) returns (3, 3), not (3,). Which rule produced that?

4.1.Q3 Structure — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

4.1.Q4 Transfer — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

4.1.Q5 Recall — What is the most important edge case in Data Preparation & Feature Engineering?

4.1.Q6 Meaning — What proves the skill is transferable?

Answer Key – Chapter 4 – Data Preparation & Feature Engineering

4.1.Q1. A boolean mask: the array must be dtype bool — np.array([False, True, False, True]) (or a comparison).

4.1.Q2. Broadcasting right-aligned (1,3) with (3,1): both stretch — an outer sum. Reshape one operand to match intent.

4.1.Q3. A — Data preparation is where projects are won: clean, scale, encode and split before training.

4.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

4.1.Q5. Fit scalers on training data only; otherwise validation data leaks into the model.

4.1.Q6. Split a tiny table into train/test first, then compute the mean only on train.

Chapter 5 · Evaluation Metrics from Scratch

Accuracy, precision, recall, F1, MSE, AUC — computed by hand.

Chapter Skill Map

What this chapter adds: In Evaluation Metrics from Scratch, the focus is accuracy, precision, recall, f1, mse, auc — computed by hand. You practise it across 9 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 5 assumes the reading habits from Chapter 4: Data Preparation & Feature Engineering. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 6, Classical ML with scikit-learn, reuses this chapter's discipline with a new kind of problem.

1 **Projects 65-67**
Recognise the pattern and read the code slowly. *name it first*

2 **Projects 68-70**
Trace similar programs with less prompting and explain what changed.

3 **Projects 71-73**
Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Evaluation Metrics from Scratch: what problem does this solve for a working programmer?

name the move before typing

Metrics define what success means: accuracy, precision, recall, F1 and error each answer a different question.

This chapter's focus is: Accuracy, precision, recall, F1, MSE, AUC — computed by hand. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
y_true = [1, 0, 1, 1]
y_pred = [1, 0, 0, 1]
tp = sum(t == p == 1 for t, p in zip(y_true, y_pred))
fp = sum(t == 0 and p == 1 for t, p in zip(y_true, y_pred))
fn = sum(t == 1 and p == 0 for t, p in zip(y_true, y_pred))
precision = tp / (tp + fp)
recall = tp / (tp + fn)
print(f"precision={precision:.2f} recall={recall:.2f}")
```

```
output
precision=1.00 recall=0.67
```

The model made no false positives, but it missed one real positive. *predict first, verify second*

TRUE POSITIVE

A positive case correctly predicted positive.

PRECISION

Of predicted positives, the fraction that were truly positive.

RECALL

Of actual positives, the fraction the model found.

F1

The harmonic mean of precision and recall.

Beginner model

At beginner level, Evaluation Metrics from Scratch should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Metric choice is product choice: medical screening, spam filtering and ranking optimise different risks.

The edge case to respect

Class imbalance can make accuracy look good while the model fails the minority class.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Add one false positive and recompute precision by hand.

Example 1

Trace the opening Evaluation Metrics from Scratch example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read Evaluation Metrics from Scratch as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Add one false positive and recompute precision by hand.

⚠ Common Mistakes

- Copying the surface syntax for Evaluation Metrics from Scratch without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Class imbalance can make accuracy look good while the model fails the minority class.
- Trusting the output before predicting it by hand.

This chapter contains 9 hands-on mini-projects (Projects 65–73). Type each one out, run it, and experiment.

Project 65 Classification Accuracy

💡 The Idea Behind It

You can't improve what you don't measure. These are the metrics that tell you whether a model is any good.

The deeper point is learning how Python turns a small intention like classification accuracy into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 66 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

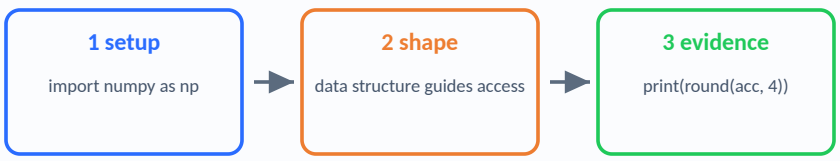
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 66 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(round(acc, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`y_true = np.array([1, 0, 1, 1, 0, 1])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(acc, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `y_true = np.array([1, 0, 1, 1, 0, 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. 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

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	Classification Accuracy: connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, y_true, array, y_pred. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y_true = np.array([1, 0, 1, 1, 0, 1]) | Evidence: print(round(acc, 4))

Reader to Writer Bridge

1. Read: explain classification accuracy 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 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(round(acc, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of classification accuracy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Accuracy is the fraction of predictions that match the labels.

```
import numpy as np

y_true = np.array([1, 0, 1, 1, 0, 1])
y_pred = np.array([1, 0, 0, 1, 0, 1])
acc = (y_true == y_pred).mean()
print(round(acc, 4))
```

```
output
0.8333
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(acc, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.

2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add a second metric for comparison.

⚠ Common Mistakes

- Confusing precision with recall.
- Forgetting to scale features for distance-based models (kNN, SVM).
- Letting class imbalance pick the majority class — stratify the split.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 66 Mean Squared Error

The Idea Behind It

You can't improve what you don't measure. These are the metrics that tell you whether a model is any good. The deeper point is learning how Python turns a small intention like mean squared error into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 65, where you practised classification accuracy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 67 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

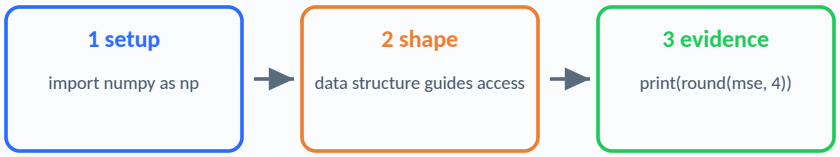
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 65: classification accuracy. Project 67 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(round(mse, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`y = np.array([3.0, 5.0, 2.5, 7.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(mse, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `y = np.array([3.0, 5.0, 2.5, 7.0])`

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. 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	Mean Squared Error: connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, array, pred, mse. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([3.0, 5.0, 2.5, 7.0]) | Evidence: print(round(mse, 4))

🛠️ Reader to Writer Bridge

1. Read: explain mean squared 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 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(round(mse, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of mean squared 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)

MSE averages the squared gap between prediction and truth.

```
import numpy as np

y = np.array([3.0, 5.0, 2.5, 7.0])
pred = np.array([2.8, 5.2, 2.0, 6.5])
mse = ((y - pred) ** 2).mean()
print(round(mse, 4))
```

```
output
0.145
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(mse, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Compute the metric on your own arrays.

⚠ Common Mistakes

- Reporting accuracy on imbalanced data.
- Tuning hyperparameters against the test set (it becomes a second train set).
- Reading feature_importances_ as causality.

Skills gained choosing and reading data structures predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 67 Precision, Recall and F1

The Idea Behind It

You can't improve what you don't measure. These are the metrics that tell you whether a model is any good. The deeper point is learning how Python turns a small intention like precision, recall and f1 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 66, where you practised mean squared error, and asks you to reuse the same reasoning pattern. reuse the pattern Project 68 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 66: mean squared error. Project 68 changes the surface problem while preserving the underlying method. progress check

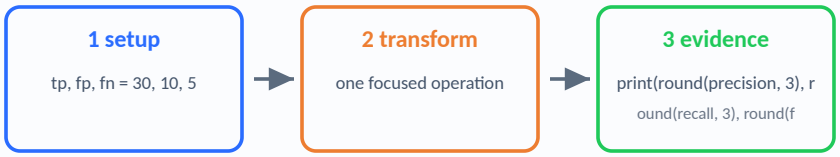
Reading target: precision, recall and f1 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `tp, fp, fn = 30, 10, 5` 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: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(round(precision, 3), round(recall, 3), round(f1, 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, 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	`tp, fp, fn = 30, 10, 5`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`precision = tp / (tp + fp)`	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(round(precision, 3), round(recall, 3), round(f1, 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `precision = tp / (tp + fp)`

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.

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	Precision, Recall and F1: connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: tp, fp, fn, precision, recall. 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: tp, fp, fn = 30, 10, 5 | Core: precision = tp / (tp + fp) | Evidence: print(round(precision, 3), round(recall, 3), round(f1, 3))

🛠️ Reader to Writer Bridge

1. Read: explain precision, recall and f1 in one sentence before touching the 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(round(precision, 3), round(recall, 3), round(f1, 3))`. Then compare punctuation, spacing, and order.

   prediction before proof
2. Explain: point to `tp, fp, fn = 30, 10, 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 precision, recall and f1 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

F1 is the harmonic mean of precision and recall.

```
tp, fp, fn = 30, 10, 5
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f1 = 2 * precision * recall / (precision + recall)
print(round(precision, 3), round(recall, 3), round(f1, 3))
```

```
output
0.75 0.857 0.8
```

How it works:

1. Start at `tp, fp, fn = 30, 10, 5`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(precision, 3), round(recall, 3), round(f1, 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add a second metric for comparison.

⚠ Common Mistakes

- Confusing precision with recall.
- Evaluating on training data and celebrating.
- Comparing models trained on different splits.

Skills gained predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 68 Pearson Correlation

The Idea Behind It

You can't improve what you don't measure. These are the metrics that tell you whether a model is any good.

The deeper point is learning how Python turns a small intention like pearson correlation into a readable

sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 67, where you practised precision, recall and f1, and asks you to reuse the same reasoning pattern. reuse the pattern Project 69 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

🧭 Skill Roadmap

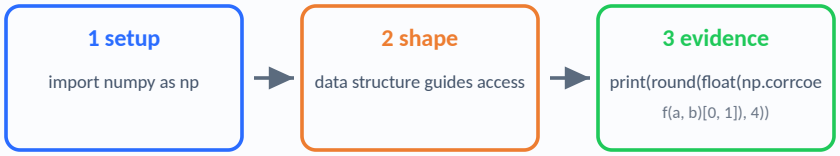
Skill step: moves you from recognition toward independent tracing. It builds on Project 67: precision, recall and f1. Project 69 changes the surface problem while preserving the underlying method. progress check

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(round(float(np.corrcoef(a, b)[0, 1]), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`a = np.array([1.0, 2.0, 3.0, 4.0, 5.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(np.corrcoef(a, b)[0, 1]), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Evaluation Metrics from Scratch becomes practical when you can build the syntax 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 = np.array([1.0, 2.0, 3.0, 4.0, 5.0])`

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. 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	Pearson Correlation: connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, array, round, corrcoef. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([1.0, 2.0, 3.0, 4.0, 5.0]) | Evidence: print(round(float(np.corrcoef(a, b)[0, 1]), 4))

Reader to Writer Bridge

1. Read: explain pearson correlation in one sentence before touching the 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(round(float(np.corrcoef(a, b)[0, 1]), 4))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of pearson correlation and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

From Concept to Code (Python)

corrcoef measures how strongly two variables move together.

```
import numpy as np

a = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
b = np.array([2.0, 4.0, 5.0, 4.0, 5.0])
print(round(float(np.corrcoef(a, b)[0, 1]), 4))
```

```
output
0.7746
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on.

anchor line

2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(np.corrcoef(a, b)[0, 1]), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output.

evidence line

Try it:

1. Change one input value or literal and predict the new output before running it.

2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Compute the metric on your own arrays.

⚠ Common Mistakes

- Reporting accuracy on imbalanced data.
- Ignoring random_state and wondering why results wander.
- Forgetting to scale features for distance-based models (kNN, SVM).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 69 Column Sums of a 7x7 Matrix (set 55)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 68, where you practised pearson correlation, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 70 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 68: pearson correlation. Project 70 changes the surface problem while preserving the underlying method.

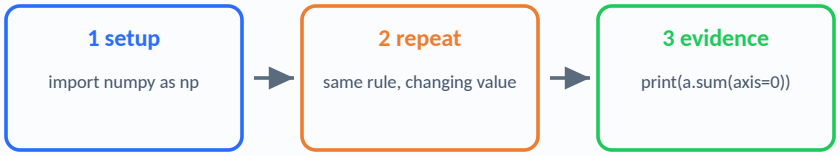
progress check

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Evaluation Metrics from Scratch becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 55): connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Refitting the vectoriser/scaler per fold inconsistently.
- Tuning hyperparameters against the test set (it becomes a second train set).

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 70 Summarise arange 1..15

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..15 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 69, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 71 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 69: column sums of a 7x7 matrix. Project 71 changes the surface problem while preserving the underlying method.

progress check

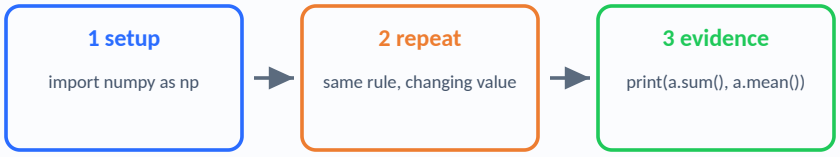
Reading target: summarise arange 1..15 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.

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

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`a = np.arange(1, 16)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Evaluation Metrics from Scratch becomes practical when you can build the syntax 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 = np.arange(1, 16)`

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. 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	Summarise arange 1..15: connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 16) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..15 in one sentence before touching the 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, Project 71 should feel like a variation, 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.sum(), a.mean())`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import numpy as np` and say why 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 arange 1..15 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```
import numpy as np

a = np.arange(1, 16)
print(a.sum(), a.mean())
```

```
output
120 8.0
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Letting class imbalance pick the majority class — stratify the split.
- Evaluating on training data and celebrating.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 71 Sort an Array (set 57)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 70, where you practised summarise arange 1..15, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 72 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

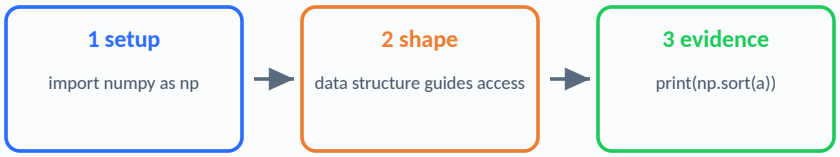
Skill step: pushes the same idea into a less automatic situation. It builds on Project 70: summarise arange 1..15. Project 72 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`a = np.array([33, 44, 55, 66, 77])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Evaluation Metrics from Scratch becomes practical when you can build the syntax 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 = np.array([33, 44, 55, 66, 77])`

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. 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	Sort an Array (set 57): connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([33, 44, 55, 66, 77]) | Evidence: print(np.sort(a))

Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 70.

read with a pencil

3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```
import numpy as np

a = np.array([33, 44, 55, 66, 77])
print(np.sort(a))
```

```
output
[33 44 55 66 77]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Reading feature_importances_ as causality.
- Ignoring random_state and wondering why results wander.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 72 Scale arange 1..5 by 5

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..5 by 5 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 71, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 73 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 71: sort an array. Project 73 changes the surface problem while preserving the underlying method. *progress check*

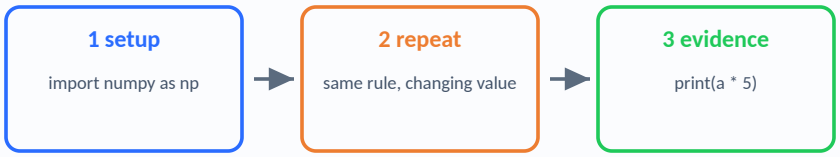
Reading target: scale arange 1..5 by 5 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(a * 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`a = np.arange(1, 6)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a * 5)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Evaluation Metrics from Scratch becomes practical when you can build the syntax 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 = np.arange(1, 6)`

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. 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	Scale arange 1..5 by 5: connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 6) | Evidence: print(a * 5)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..5 by 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 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(a * 5)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..5 by 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)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np

a = np.arange(1, 6)
print(a * 5)
```

```
output
[ 5 10 15 20 25]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 5)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Comparing models trained on different splits.
- Refitting the vectoriser/scaler per fold inconsistently.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 73 Boolean Mask: Above Mean (set 59)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Evaluation Metrics from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 72, where you practised scale arange 1..5 by 5, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

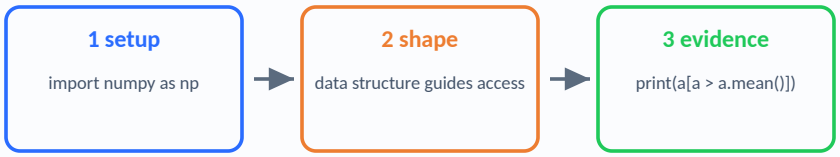
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 72: scale arange 1..5 by 5. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: accuracy, precision, recall, f1, mse, auc — computed by hand.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: accuracy, precision, recall, f1, mse, auc — computed by hand.
2. Apply the rule	`a = np.array([24, 29, 34, 39, 44, 49])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Evaluation Metrics from Scratch becomes practical when you can build the syntax 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 = np.array([24, 29, 34, 39, 44, 49])`

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. 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	Boolean Mask: Above Mean (set 59): connect the surface task to the chapter skill, accuracy, precision, recall, f1, mse, auc — computed by hand.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([24, 29, 34, 39, 44, 49]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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, 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([24, 29, 34, 39, 44, 49])
print(a[a > a.mean()])
```

```
output
[39 44 49]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Forgetting to scale features for distance-based models (kNN, SVM).
- Letting class imbalance pick the majority class — stratify the split.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 9 mini-projects, Evaluation Metrics from Scratch should feel like a reusable Python move: accuracy, precision, recall, f1, mse, auc — computed by hand. *chapter checkpoint*

1 Read
Open any project from Projects 65-73 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 (Classical ML with scikit-learn) 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

Recall · Meaning · Structure · Transfer

5.1.Q1 Recall Turn back to Project 67 (Precision, Recall and F1). Without running it, what exactly does it print?

5.1.Q2 Meaning Bug hunt: GridSearch picked parameters using the test set, and the final "test" score is glowing. What is the honest protocol?

5.1.Q3 Structure Bug hunt: kNN scores 0.93 on raw features but 0.97 after one preprocessing step. Which step and why?

5.1.Q4 Meaning What is the main problem this chapter's topic solves?

- A It gives a named, repeatable way to handle the chapter's programming situation.
- B It removes the need to test code.
- C It makes every program shorter.
- D It only matters for memorising syntax.

5.1.Q5 Recall What should you identify before running a project?

- A The setup, the rule or transformation, and the evidence line.
- B Only the final answer.
- C Only the imports.
- D Only the file name.

5.1.Q6 Meaning What is the most important edge case in Evaluation Metrics from Scratch?

5.1.Q7 Structure What proves the skill is transferable?

Answer Key – Chapter 5 – Evaluation Metrics from Scratch

5.1.Q1. `0.75 0.857 0.8`

5.1.Q2. Three-way separation: tune with CV on the training data only; touch the test set once, at the very end.

5.1.Q3. Scaling — kNN distances are dominated by large-range features until all are standardised.

5.1.Q4. A — Metrics define what success means: accuracy, precision, recall, F1 and error each answer a different question.

5.1.Q5. A — prediction comes from setup, rule and evidence, not from guessing.

5.1.Q6. Class imbalance can make accuracy look good while the model fails the minority class.

5.1.Q7. Add one false positive and recompute precision by hand.



PART II

Intermediate Skills

With the basics in hand, take on functions, data structures, recursion, algorithms and object-oriented design.

Chapter 6 – Classical ML with scikit-learn · Chapter 7 – Regression & Classification from Scratch · Chapter 8 – Optimization & Gradient Descent · Chapter 9 – Neural Networks from Scratch · Chapter 10 – PyTorch Fundamentals

Chapter 6 · Classical ML with scikit-learn

Fit/predict pipelines, trees, ensembles, cross-validation.

Chapter Skill Map

What this chapter adds: In Classical ML with scikit-learn, the focus is fit/predict pipelines, trees, ensembles, cross-validation. You practise it across 11 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 6 assumes the reading habits from Chapter 5: Evaluation Metrics from Scratch. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 7, Regression & Classification from Scratch, reuses this chapter's discipline with a new kind of problem.

1 **Projects 74-76**
Recognise the pattern and read the code slowly. *name it first*

2 **Projects 77-79**
Trace similar programs with less prompting and explain what changed.

3 **Projects 80-84**
Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Classical ML with scikit-learn: what problem does this solve for a working programmer?

name the move before typing

scikit-learn turns model work into a fit/predict pipeline with repeatable preprocessing and evaluation.

This chapter's focus is: Fit/predict pipelines, trees, ensembles, cross-validation. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
from sklearn.tree import DecisionTreeClassifier

X = [[0, 0], [1, 1], [1, 0], [0, 1]]
y = [0, 1, 1, 0]
model = DecisionTreeClassifier(random_state=0).fit(X, y)
print(model.predict([[1, 1]]).tolist())
```

```
output
[1]
```

fit learns a rule from X and y; predict applies it to a new row. *predict first, verify second*

FIT

The training step that learns parameters from data.

PREDICT

The inference step that produces outputs for new examples.

PIPELINE

A repeatable chain of preprocessing and model steps.

CROSS-VALIDATION

Repeated train/validation splits used to estimate generalisation.

Beginner model

At beginner level, Classical ML with scikit-learn should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

The mature workflow compares baselines, validates honestly and saves the exact pipeline.

The edge case to respect

Randomness must be controlled with random_state when you want reproducible learning examples.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Replace the tree with a logistic regression and compare the prediction.

Example 1

Trace the opening Classical ML with scikit-learn example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read Classical ML with scikit-learn as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Replace the tree with a logistic regression and compare the prediction.

⚠ Common Mistakes

- Copying the surface syntax for Classical ML with scikit-learn without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Randomness must be controlled with random_state when you want reproducible learning examples.
- Trusting the output before predicting it by hand.

This chapter contains 11 hands-on mini-projects (Projects 74–84). Type each one out, run it, and experiment.

Project 74 Linear Regression

💡 The Idea Behind It

scikit-learn is the standard toolkit for classical machine learning — a consistent fit/predict API across dozens of models. The deeper point is learning how Python turns a small intention like linear regression into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen `code reading`: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, `Classical ML with scikit-learn`, to a concrete mini-program.
- Read the program as `input or setup`, `transformation`, then `output`. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 75 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

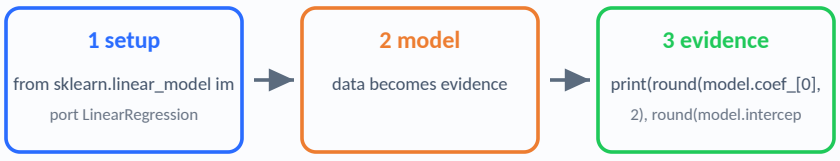
`Skill step:` opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 75 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- `Setup:` `from sklearn.linear_model import LinearRegression` 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: fit/predict pipelines, trees, ensembles, cross-validation.
- `Evidence:` `print(round(model.coef_[0], 2), round(model.intercept_, 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, `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	`from sklearn.linear_model import LinearRegression`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`model = LinearRegression().fit(X, y)`	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(round(model.coef_[0], 2), round(model.intercept_, 2))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `model = LinearRegression().fit(X, y)`

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. 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	Linear Regression: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: sklearn, linear_model, LinearRegression, numpy, np. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: from sklearn.linear_model import LinearRegression | Core: model = LinearRegression().fit(X, y) | Evidence: print(round(model.coef_[0], 2), round(model.intercept_, 2))

Reader to Writer Bridge

1. Read: explain linear regression 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 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(round(model.coef_[0], 2), round(model.intercept_, 2))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from sklearn.linear_model import LinearRegression` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of linear regression and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

fit learns the slope and intercept of the best-fit line.

```python
from sklearn.linear_model import LinearRegression
import numpy as np

X = np.array([[1], [2], [3], [4]])
y = np.array([2, 4, 6, 8])
model = LinearRegression().fit(X, y)
print(round(model.coef_[0], 2), round(model.intercept_, 2))
```

```
output
2.0 0.0
```

How it works:

1. Start at `from sklearn.linear_model import LinearRegression`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(model.coef_[0], 2), round(model.intercept_, 2))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Swap in a different estimator.

⚠ Common Mistakes

- Scaling before splitting causes data leakage.
- Forgetting that groupby drops NaN keys silently.
- Using apply(row-wise) where a vectorised column op was available.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 75 Logistic Regression on Iris

The Idea Behind It

scikit-learn is the standard toolkit for classical machine learning — a consistent fit/predict API across dozens of models. The deeper point is learning how Python turns a small intention like logistic regression on iris into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 74, where you practised linear regression, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 76 will change the surface task again, so treat this one as a

checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

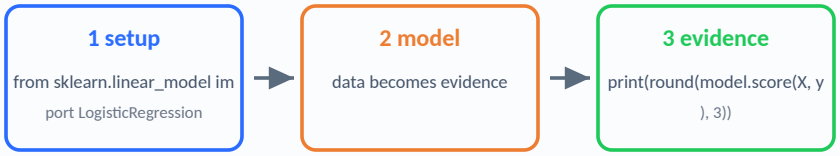
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 74: linear regression. Project 76 changes the surface problem while preserving the underlying method. *progress check*

Reading target: logistic regression on iris is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from sklearn.linear_model import LogisticRegression` 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: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(round(model.score(X, y), 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, 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	`from sklearn.linear_model import LogisticRegression`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`model = LogisticRegression(max_iter=200).fit(X, y)`	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(round(model.score(X, y), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `model = LogisticRegression(max_iter=200).fit(X, y)`

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. 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	Logistic Regression on Iris: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: sklearn, linear_model, LogisticRegression, datasets, load_iris. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: from sklearn.linear_model import LogisticRegression | Core: model = LogisticRegression(max_iter=200).fit(X, y) | Evidence: print(round(model.score(X, y), 3))

🛠️ Reader to Writer Bridge

1. Read: explain logistic regression on iris in one sentence before touching the 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(round(model.score(X, y), 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from sklearn.linear_model import LogisticRegression` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of logistic regression on iris and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

score returns accuracy on the given data.

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

X, y = load_iris(return_X_y=True)
model = LogisticRegression(max_iter=200).fit(X, y)
print(round(model.score(X, y), 3))
```

```
output
0.973
```

How it works:

1. Start at `from sklearn.linear_model import LogisticRegression`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(model.score(X, y), 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Print a full classification report.

⚠ Common Mistakes

- Judging a model on its training accuracy alone.
- Chained indexing df[a][b] = x — the SettingWithCopy trap; use .loc.
- Parsing dates as strings and sorting alphabetically.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 76 Decision Tree with a Train/Test Split

The Idea Behind It

scikit-learn is the standard toolkit for classical machine learning — a consistent fit/predict API across dozens of models. The deeper point is learning how Python turns a small intention like decision tree with a train/test split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 75, where you practised logistic regression on iris, and asks you to reuse the same reasoning pattern. reuse the pattern Project 77 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 75: logistic regression on iris. Project 77 changes the surface problem while preserving the underlying method.

progress check

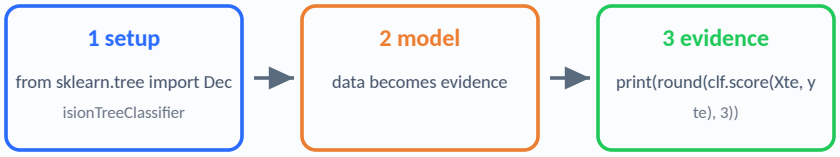
Reading target: decision tree with a train/test split is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `from sklearn.tree import DecisionTreeClassifier` 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: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(round(clf.score(Xte, yte), 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, 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	`from sklearn.tree import DecisionTreeClassifier`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`clf = DecisionTreeClassifier(max_depth=3).fit(Xtr, ytr)`	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(round(clf.score(Xte, yte), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `clf = DecisionTreeClassifier(max_depth=3).fit(Xtr, ytr)`

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. 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	Decision Tree with a Train/Test Split: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: sklearn, tree, DecisionTreeClassifier, datasets, load_iris. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: from sklearn.tree import DecisionTreeClassifier | Core: clf = DecisionTreeClassifier(max_depth=3).fit(Xtr, ytr) | Evidence: print(round(clf.score(Xte, yte), 3))

🛠️ Reader to Writer Bridge

1. Read: explain decision tree with a train/test split in one sentence before touching the 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(round(clf.score(Xte, yte), 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from sklearn.tree import DecisionTreeClassifier` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of decision tree with a train/test split and name what must stay the 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 held-out test set estimates real-world accuracy.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
```

```
from sklearn.model_selection import train_test_split

X, y = load_iris(return_X_y=True)
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.3, random_state=0)
clf = DecisionTreeClassifier(max_depth=3).fit(Xtr, ytr)
print(round(clf.score(Xte, yte), 3))
```

```
output
0.978
```

How it works:

1. Start at `from sklearn.tree import DecisionTreeClassifier`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(clf.score(Xte, yte), 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Swap in a different estimator.

⚠ Common Mistakes

- Scaling before splitting causes data leakage.
- Resetting vs keeping the index after groupby without noticing.
- Aggregating the label column away and joining it back wrong.

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 77 K-Means Clustering

The Idea Behind It

scikit-learn is the standard toolkit for classical machine learning — a consistent fit/predict API across dozens of models. The deeper point is learning how Python turns a small intention like k-means clustering into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 76, where you practised decision tree with a train/test split, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 78 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 76: decision tree with a train/test split. Project 78 changes the surface problem while preserving the underlying method.

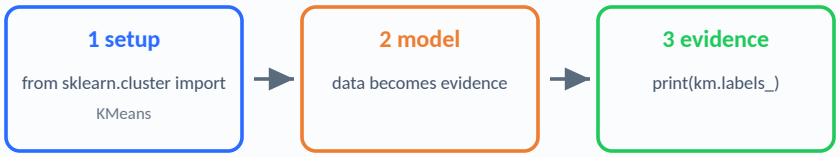
progress check

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

Logic Lens

- Setup: `from sklearn.cluster import KMeans` 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: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(km.labels_)` exposes the result; match it 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	`from sklearn.cluster import KMeans`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`km = KMeans(n_clusters=2, n_init=10, random_state=0).fit(X)`	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(km.labels_)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `km = KMeans(n_clusters=2, n_init=10, random_state=0).fit(X)`

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. 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	K-Means Clustering: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.

Layer	What to read
Names	Watch these names first: sklearn, cluster, KMeans, numpy, np. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: from sklearn.cluster import KMeans | Core: km = KMeans(n_clusters=2, n_init=10, random_state=0).fit(X) | Evidence: print(km.labels_)

🛠️ Reader to Writer Bridge

1. Read: explain k-means clustering in one sentence before touching the 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(km.labels_)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from sklearn.cluster import KMeans` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of k-means clustering and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

K-Means groups points into k clusters by proximity.

```python
from sklearn.cluster import KMeans
import numpy as np

X = np.array([[1, 1], [1.5, 2], [8, 8], [9, 8]])
km = KMeans(n_clusters=2, n_init=10, random_state=0).fit(X)
print(km.labels_)
```

```
output
[1 1 0 0]
```

How it works:

1. Start at `from sklearn.cluster import KMeans`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(km.labels_)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Print a full classification report.

⚠ Common Mistakes

- Judging a model on its training accuracy alone.
- Modifying a slice and expecting the parent DataFrame to change.
- Forgetting that groupby drops NaN keys silently.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 78 Random Forest with Cross-Validation

The Idea Behind It

scikit-learn is the standard toolkit for classical machine learning — a consistent fit/predict API across dozens of models. The deeper point is learning how Python turns a small intention like random forest with cross-validation into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 77, where you practised k-means clustering, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 79 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

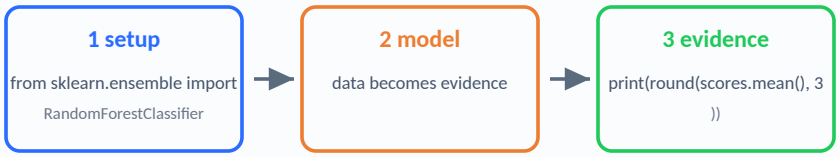
Skill step: moves you from recognition toward independent tracing. It builds on Project 77: k-means clustering. Project 79 changes the surface problem while preserving the underlying method. *progress check*

Reading target: random forest with cross-validation is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from sklearn.ensemble import RandomForestClassifier` 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: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(round(scores.mean(), 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, 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	`from sklearn.ensemble import RandomForestClassifier`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`from sklearn.datasets import load_wine`	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(round(scores.mean(), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax 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 sklearn.datasets import load_wine`

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. 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	Random Forest with Cross-Validation: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: sklearn, ensemble, RandomForestClassifier, datasets, load_wine. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: from sklearn.ensemble import RandomForestClassifier | Core: from sklearn.datasets import load_wine | Evidence: print(round(scores.mean(), 3))

Reader to Writer Bridge

1. Read: explain random forest with cross-validation in one sentence before touching the 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(round(scores.mean(), 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from sklearn.ensemble import RandomForestClassifier` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of random forest with cross-validation and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cross-validation averages accuracy over several folds.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_wine
from sklearn.model_selection import cross_val_score

X, y = load_wine(return_X_y=True)
clf = RandomForestClassifier(n_estimators=100, random_state=0)
scores = cross_val_score(clf, X, y, cv=5)
print(round(scores.mean(), 3))
```

```
output
0.983
```

How it works:

1. Start at `from sklearn.ensemble import RandomForestClassifier`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(scores.mean(), 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.

2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Swap in a different estimator.

⚠ Common Mistakes

- Scaling before splitting causes data leakage.
- merge on the wrong key and silently duplicating rows.
- Chained indexing df[a][b] = x — the SettingWithCopy trap; use .loc.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 79 A Scaling + SVM Pipeline

The Idea Behind It

scikit-learn is the standard toolkit for classical machine learning — a consistent fit/predict API across dozens of models. The deeper point is learning how Python turns a small intention like a scaling + svm pipeline into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 78, where you practised random forest with cross-validation, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 80 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 78: random forest with cross-validation. Project 80 changes the surface problem while preserving the underlying method.

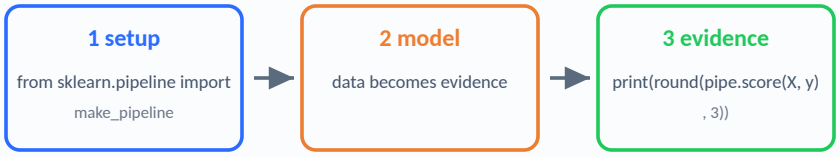
progress check

Reading target: a scaling + svm pipeline is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `from sklearn.pipeline import make_pipeline` 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: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(round(pipe.score(X, y), 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, 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	`from sklearn.pipeline import make_pipeline`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`pipe.fit(X, y)`	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(round(pipe.score(X, y), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `pipe.fit(X, y)`

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. 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	A Scaling + SVM Pipeline: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: sklearn, pipeline, make_pipeline, preprocessing, StandardScaler. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: from sklearn.pipeline import make_pipeline | Core: pipe.fit(X, y) | Evidence: print(round(pipe.score(X, y), 3))

🛠️ Reader to Writer Bridge

1. Read: explain a scaling + svm pipeline in one sentence before touching the 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(round(pipe.score(X, y), 3))`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `from sklearn.pipeline import make_pipeline` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of a scaling + svm pipeline and name what must stay the 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 pipeline chains preprocessing and model into one estimator.

```python
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.datasets import load_iris

X, y = load_iris(return_X_y=True)
pipe = make_pipeline(StandardScaler(), SVC())
pipe.fit(X, y)
print(round(pipe.score(X, y), 3))
```

```
output
0.973
```

How it works:

1. Start at `from sklearn.pipeline import make_pipeline`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(pipe.score(X, y), 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Print a full classification report.

⚠ Common Mistakes

- Judging a model on its training accuracy alone.
- Using apply(row-wise) where a vectorised column op was available.
- Resetting vs keeping the index after groupby without noticing.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 80 Column Sums of a 7x7 Matrix (set 60)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 79, where you practised a scaling + svm pipeline, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 81 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 79: a scaling + svm pipeline. Project 81 changes the surface problem while preserving the underlying method. *progress check*

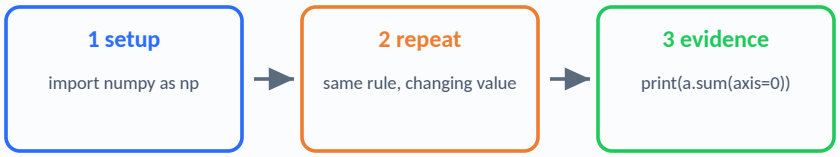
Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.

- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 60): connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```python
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Parsing dates as strings and sorting alphabetically.
- Modifying a slice and expecting the parent DataFrame to change.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 81 Summarise arange 1..16

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..16 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 80, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 82 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

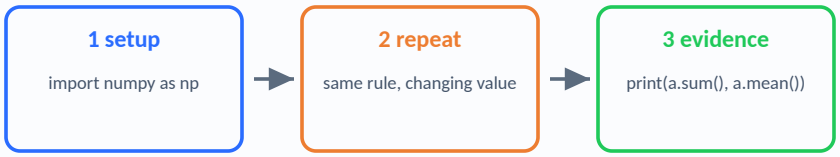
Skill step: pushes the same idea into a less automatic situation. It builds on Project 80: column sums of a 7x7 matrix. Project 82 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..16 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`a = np.arange(1, 17)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax 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 = np.arange(1, 17)`

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. 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	Summarise arange 1..16: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 17) | Evidence: print(a.sum(), a.mean())

Reader to Writer Bridge

1. Read: explain summarise arange 1..16 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..16 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 17)
print(a.sum(), a.mean())
```

```
output
136 8.5
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Aggregating the label column away and joining it back wrong.
- merge on the wrong key and silently duplicating rows.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 82 Sort an Array (set 62)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 81, where you practised summarise arange 1..16, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 83 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

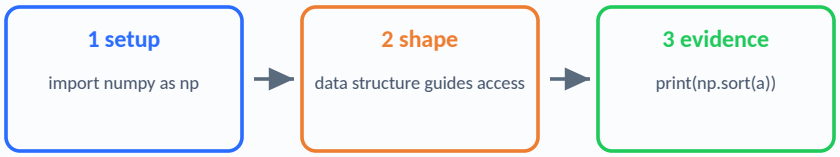
Skill step: pushes the same idea into a less automatic situation. It builds on Project 81: summarise arange 1..16. Project 83 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(np.sort(a))` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`a = np.array([68, 79, 90, 11, 22])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax 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 = np.array([68, 79, 90, 11, 22])`

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. 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	Sort an Array (set 62): connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([68, 79, 90, 11, 22]) | Evidence: print(np.sort(a))

🛠️ Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```python
import numpy as np

a = np.array([68, 79, 90, 11, 22])
print(np.sort(a))
```

```
output
[11 22 68 79 90]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Forgetting that groupby drops NaN keys silently.
- Using apply(row-wise) where a vectorised column op was available.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 83 Scale arange 1..6 by 10

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..6 by 10 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 82, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 84 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

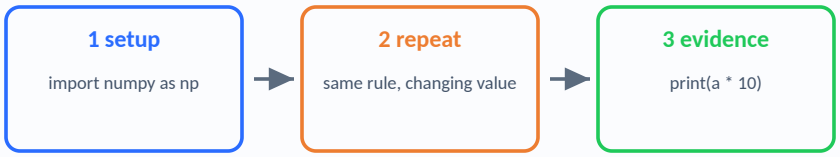
Skill step: pushes the same idea into a less automatic situation. It builds on Project 82: sort an array. Project 84 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..6 by 10 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to write one rule and let repetition do the volume work. It adds that move to the chapter skill: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(a * 10)` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`a = np.arange(1, 7)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(a * 10)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax 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 = np.arange(1, 7)`

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. 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	Scale arange 1..6 by 10: connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 7) | Evidence: print(a * 10)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..6 by 10 in one sentence before touching the 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(a * 10)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..6 by 10 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```python
import numpy as np

a = np.arange(1, 7)
print(a * 10)
```

```
output
[10 20 30 40 50 60]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a * 10)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Chained indexing df[a][b] = x — the SettingWithCopy trap; use .loc.
- Parsing dates as strings and sorting alphabetically.

Skills gained tracing repeated work | choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 84 Boolean Mask: Above Mean (set 64)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Classical ML with scikit-learn, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.

- It builds on Project 83, where you practised scale arange 1..6 by 10, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

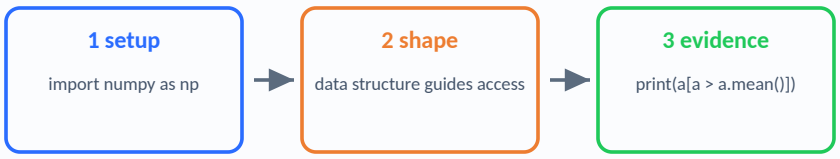
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 83: scale arange 1..6 by 10. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: fit/predict pipelines, trees, ensembles, cross-validation.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: fit/predict pipelines, trees, ensembles, cross-validation.
2. Apply the rule	`a = np.array([39, 44, 49, 4, 9, 14])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Classical ML with scikit-learn becomes practical when you can build the syntax 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 = np.array([39, 44, 49, 4, 9, 14])`

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. 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	Boolean Mask: Above Mean (set 64): connect the surface task to the chapter skill, fit/predict pipelines, trees, ensembles, cross-validation.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([39, 44, 49, 4, 9, 14]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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, 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([39, 44, 49, 4, 9, 14])
print(a[a > a.mean()])
```

```
output
[39 44 49]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Resetting vs keeping the index after groupby without noticing.
- Aggregating the label column away and joining it back wrong.

Skills gained choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Chapter Progress Checkpoint

What this chapter added: after 11 mini-projects, Classical ML with scikit-learn should feel like a reusable Python move: fit/predict pipelines, trees, ensembles, cross-validation. *chapter checkpoint*

1 Read
Open any project from Projects 74-84 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 (Regression & Classification from Scratch) 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

Recall · Meaning · Structure · Transfer

6.1.Q1 Recall — Bug hunt: `df[df.score > 80]["passed"] = True` warns and changes nothing. What is the idiom?

6.1.Q2 Meaning — Bug hunt: after a merge, the row count doubled. What property of the key explains it?

6.1.Q3 Structure — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

6.1.Q4 Transfer What should you identify before running a project?

A The setup, the rule or transformation, and the evidence line.

B Only the final answer.

C Only the imports.

D Only the file name.

6.1.Q5 Recall What is the most important edge case in Classical ML with scikit-learn?

6.1.Q6 Transfer What proves the skill is transferable?

Answer Key – Chapter 6 – Classical ML with scikit-learn

6.1.Q1. df.loc[df.score > 80, "passed"] = True — one indexer, one assignment, no copy ambiguity.

6.1.Q2. The key was not unique on one side — many-to-many matches multiply rows; check with.duplicated() first.

6.1.Q3. A — scikit-learn turns model work into a fit/predict pipeline with repeatable preprocessing and evaluation.

6.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

6.1.Q5. Randomness must be controlled with random_state when you want reproducible learning examples.

6.1.Q6. Replace the tree with a logistic regression and compare the prediction.

Chapter 7 · Regression & Classification from Scratch

Linear models, logistic regression and their loss functions, in NumPy.

Chapter Skill Map

What this chapter adds: In Regression & Classification from Scratch, the focus is linear models, logistic regression and their loss functions, in numpy. You practise it across 35 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 7 assumes the reading habits from Chapter 6: Classical ML with scikit-learn. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 8, Optimization & Gradient Descent, reuses this chapter's discipline with a new kind of problem.

1 **Projects 85-95**	2 **Projects 96-106**	3 **Projects 107-119**
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

Regression & Classification from Scratch: what problem does this solve for a working programmer?

name the move before typing

Building regression and classification from scratch reveals the loss functions hidden inside libraries.

This chapter's focus is: Linear models, logistic regression and their loss functions, in NumPy. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
import math

w, b, x = 2.0, -1.0, 3.0
logit = w * x + b
prob = 1 / (1 + math.exp(-logit))
print(round(prob, 3))
```

```
output
0.993
```

A large positive logit becomes a probability close to 1. *predict first, verify second*

LOGIT

The raw linear score before a sigmoid turns it into a probability.

SIGMOID

A function mapping any real value into the range 0..1.

LOSS

A number measuring how wrong a prediction is.

GRADIENT

The direction a parameter should move to reduce loss.

Beginner model

At beginner level, Regression & Classification from Scratch should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Scratch implementations build trust: you can debug the math when a library model misbehaves.

The edge case to respect

Probabilities exactly 0 or 1 break log loss, so numerical code often clips values with epsilon.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Change w to -2 and explain why the probability flips.

Example 1

Trace the opening Regression & Classification from Scratch example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read Regression & Classification from Scratch as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Change w to -2 and explain why the probability flips.

⚠ Common Mistakes

- Copying the surface syntax for Regression & Classification from Scratch without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Probabilities exactly 0 or 1 break log loss, so numerical code often clips values with epsilon.
- Trusting the output before predicting it by hand.

This chapter contains 35 hands-on mini-projects (Projects 85–119). Type each one out, run it, and experiment.

Project 85 Sigmoid of 5.0

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of 5.0 into a

readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example. first anchor Project 86 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

🧭 Skill Roadmap

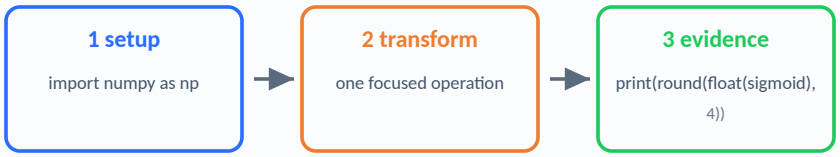
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 86 changes the surface problem while preserving the underlying method. progress check

Reading target: sigmoid of 5.0 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of 5.0: connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of 5.0 in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then 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 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of 5.0 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)

The sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = 5.0
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.9933
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Dividing by n vs n−1 in variance and wondering about the gap.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 86 Softmax of a 3-Vector (set 102)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.

- It builds on Project 85, where you practised sigmoid of 5.0, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 87 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

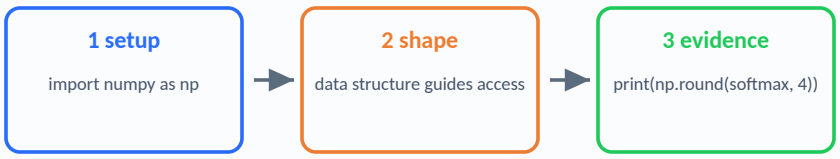
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 85: sigmoid of 5.0. Project 87 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([3.5, -3.0, -0.5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([3.5, −3.0, −0.5])`

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. 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	Softmax of a 3-Vector (set 102): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([3.5, -3.0, -0.5]) | Evidence: print(np.round(softmax, 4))

Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([3.5, -3.0, -0.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.9806 0.0015 0.018 ]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 87 Dot Product (set 103)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 86, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 88 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

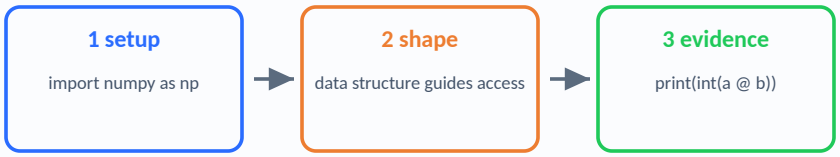
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 86: softmax of a 3-vector. Project 88 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([4, 7, 1])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([4, 7, 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. 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	Dot Product (set 103): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([4, 7, 1]) | Evidence: print(int(a @ b))

🛠️ Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([4, 7, 1])
b = np.array([7, 3, 8])
print(int(a @ b))
```

```
output
57
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Sigmoid for multi-class where softmax was required.
- Comparing float results with == instead of np.isclose in tests.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 88 Binary Cross-Entropy (set 104)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 87, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 89 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

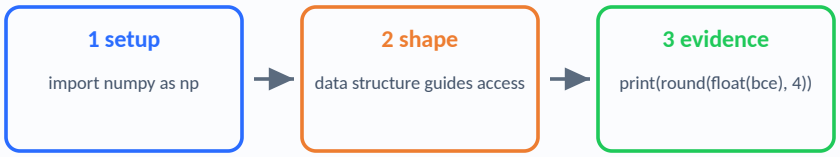
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 87: dot product. Project 89 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`y = np.array([1, 0, 1])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `y = np.array([1, 0, 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. 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	Binary Cross-Entropy (set 104): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.

Layer	What to read
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([1, 0, 1]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🤖 From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```python
import numpy as np

y = np.array([1, 0, 1])
p = np.array([0.7, 0.2, 0.5])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Mixing natural log and log2 between formula and code.
- Dividing by n vs n−1 in variance and wondering about the gap.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 89 Cosine Similarity (set 105)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 88, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 90 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

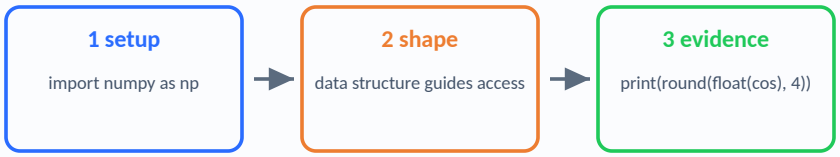
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 88: binary cross-entropy. Project 90 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([7, 2, 4], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([7, 2, 4], dtype=float)`

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. 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	Cosine Similarity (set 105): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([7, 2, 4], dtype=float) | Evidence: print(round(float(cos), 4))

Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([7, 2, 4], dtype=float)
b = np.array([5, 6, 7], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.8609
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Cosine similarity without normalising — it silently becomes a dot product.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 90 Sigmoid of -5.25

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -5.25 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 89, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 91 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

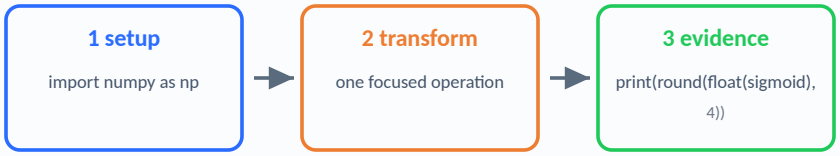
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 89: cosine similarity. Project 91 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of -5.25 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -5.25: connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of -5.25 in one sentence before touching the 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -5.25 and name what must stay the 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 sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = -5.25
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.0052
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.
- Sigmoid for multi-class where softmax was required.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 91 Cosine Similarity (set 255)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 90, where you practised sigmoid of -5.25, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 92 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

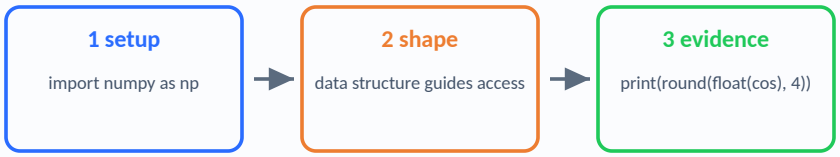
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 90: sigmoid of -5.25. Project 92 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([3, 5, 7], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([3, 5, 7], dtype=float)`

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. 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	Cosine Similarity (set 255): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.

Layer	What to read
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([3, 5, 7], dtype=float) | Evidence: print(round(float(cos), 4))

🛠️ Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([3, 5, 7], dtype=float)
b = np.array([7, 1, 2], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.5975
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).
- Mixing natural log and log2 between formula and code.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 92 Softmax of a 3-Vector (set 257)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 91, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 93 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

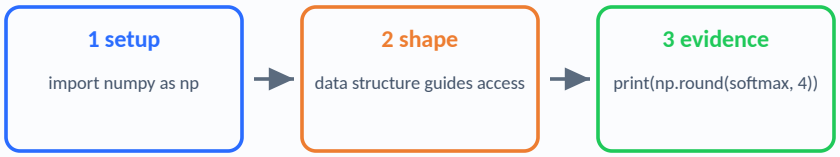
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 91: cosine similarity. Project 93 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([2.0, 4.5, -2.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([2.0, 4.5, -2.0])`

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. 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	Softmax of a 3-Vector (set 257): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([2.0, 4.5, -2.0]) | Evidence: print(np.round(softmax, 4))

Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([2.0, 4.5, -2.0])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0758 0.9229 0.0014]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Comparing float results with == instead of np.isclose in tests.
- Cosine similarity without normalising — it silently becomes a dot product.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 93 Dot Product (set 258)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 92, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 94 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

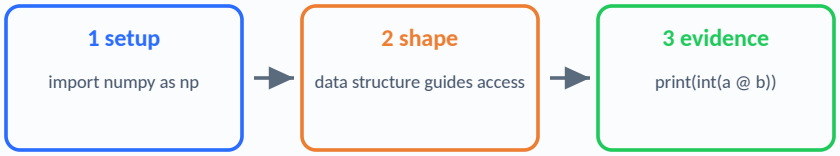
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 92: softmax of a 3-vector. Project 94 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([6, 9, 3])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([6, 9, 3])`

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. 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	Dot Product (set 258): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([6, 9, 3]) | Evidence: print(int(a @ b))

🛠️ Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([6, 9, 3])
b = np.array([2, 7, 3])
print(int(a @ b))
```

```
output
84
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Dividing by n vs n−1 in variance and wondering about the gap.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 94 Binary Cross-Entropy (set 259)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 93, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 95 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

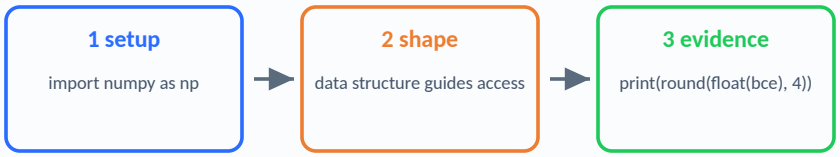
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 93: dot product. Project 95 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`y = np.array([0, 1, 0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `y = np.array([0, 1, 0])`

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. 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	Binary Cross-Entropy (set 259): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.

Layer	What to read
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([0, 1, 0]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```python
import numpy as np

y = np.array([0, 1, 0])
p = np.array([0.5, 0.8, 0.3])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 95 Cosine Similarity (set 260)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 94, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 96 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

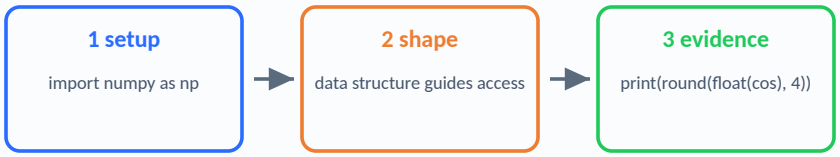
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 94: binary cross-entropy. Project 96 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([1, 3, 5], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([1, 3, 5], dtype=float)`

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. 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	Cosine Similarity (set 260): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([1, 3, 5], dtype=float) | Evidence: print(round(float(cos), 4))

Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([1, 3, 5], dtype=float)
b = np.array([1, 2, 3], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.9939
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Sigmoid for multi-class where softmax was required.
- Comparing float results with == instead of np.isclose in tests.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 96 Softmax of a 3-Vector (set 262)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 95, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 97 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

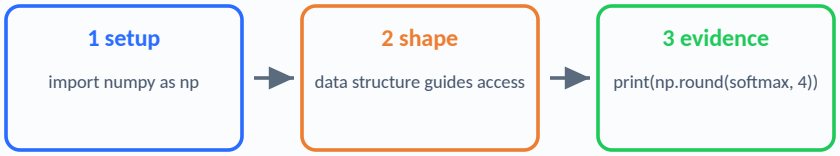
Skill step: moves you from recognition toward independent tracing. It builds on Project 95: cosine similarity. Project 97 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([0.5, 3.0, -3.5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([0.5, 3.0, -3.5])`

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. 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	Softmax of a 3-Vector (set 262): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([0.5, 3.0, -3.5]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([0.5, 3.0, -3.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0758 0.9229 0.0014]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Mixing natural log and log2 between formula and code.
- Dividing by n vs n−1 in variance and wondering about the gap.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 97 Softmax of a 3-Vector (set 177)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 96, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 98 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 96: softmax of a 3-vector. Project 98 changes the surface problem while preserving the underlying method. *progress check*

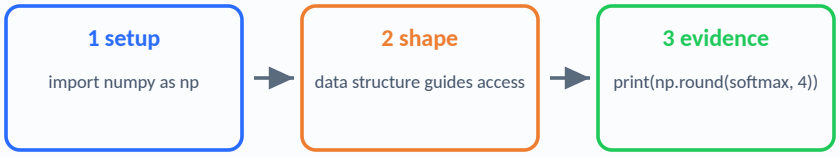
Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.

- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([-1.0, 1.5, 4.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([-1.0, 1.5, 4.0])`

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. 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	Softmax of a 3-Vector (set 177): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([-1.0, 1.5, 4.0]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([-1.0, 1.5, 4.0])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
```

```
[0.0062 0.0754 0.9184]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Cosine similarity without normalising — it silently becomes a dot product.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 98 Dot Product (set 178)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 97, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 99 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

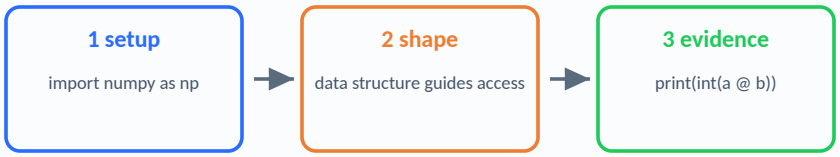
Skill step: moves you from recognition toward independent tracing. It builds on Project 97: softmax of a 3-vector. Project 99 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([7, 1, 4])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([7, 1, 4])`

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. 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	Dot Product (set 178): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([7, 1, 4]) | Evidence: print(int(a @ b))

Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([7, 1, 4])
b = np.array([4, 9, 5])
print(int(a @ b))
```

```
output
57
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.
- Sigmoid for multi-class where softmax was required.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 99 Binary Cross-Entropy (set 179)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 98, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 100 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

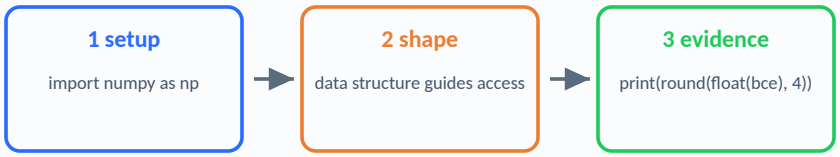
Skill step: moves you from recognition toward independent tracing. It builds on Project 98: dot product. Project 100 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`y = np.array([0, 1, 0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `y = np.array([0, 1, 0])`

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. 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	Binary Cross-Entropy (set 179): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([0, 1, 0]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```python
import numpy as np

y = np.array([0, 1, 0])
p = np.array([0.5, 0.8, 0.3])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).
- Mixing natural log and log2 between formula and code.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 100 Cosine Similarity (set 180)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 99, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 101 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 99: binary cross-entropy. Project 101 changes the surface problem while preserving the underlying method. *progress check*

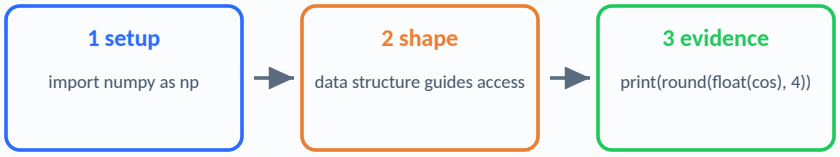
Reading target: cosine similarity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.

- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([5, 7, 2], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([5, 7, 2], dtype=float)`

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. 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	Cosine Similarity (set 180): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([5, 7, 2], dtype=float) | Evidence: print(round(float(cos), 4))

🛠️ Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```python
import numpy as np

a = np.array([5, 7, 2], dtype=float)
b = np.array([6, 7, 1], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.989
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Comparing float results with == instead of np.isclose in tests.
- Cosine similarity without normalising — it silently becomes a dot product.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 101 Sigmoid of -3.0

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -3.0 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 100, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 102 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

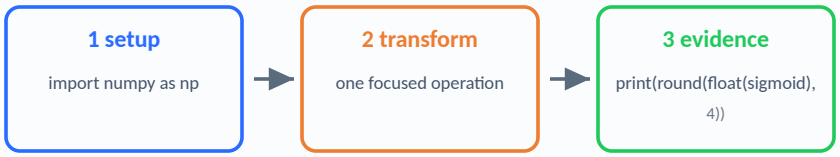
Skill step: moves you from recognition toward independent tracing. It builds on Project 100: cosine similarity. Project 102 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of -3.0 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -3.0: connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

Reader to Writer Bridge

1. Read: explain sigmoid of -3.0 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 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -3.0 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)

The sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = -3.0
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.0474
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Dividing by n vs n−1 in variance and wondering about the gap.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 102 Softmax of a 3-Vector (set 182)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 101, where you practised sigmoid of -3.0, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 103 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

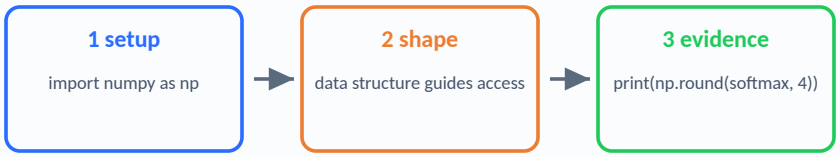
Skill step: moves you from recognition toward independent tracing. It builds on Project 101: sigmoid of -3.0. Project 103 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([-2.5, 0.0, 2.5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([-2.5, 0.0, 2.5])`

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. 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	Softmax of a 3-Vector (set 182): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([-2.5, 0.0, 2.5]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([-2.5, 0.0, 2.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0062 0.0754 0.9184]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 103 Cosine Similarity (set 25)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 102, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 104 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 102: softmax of a 3-vector. Project 104 changes the surface problem while preserving the underlying method. *progress check*

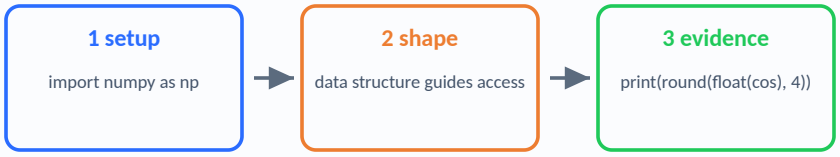
Reading target: cosine similarity is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.

- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([4, 6, 1], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([4, 6, 1], dtype=float)`

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. 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	Cosine Similarity (set 25): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([4, 6, 1], dtype=float) | Evidence: print(round(float(cos), 4))

🛠️ Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```python
import numpy as np

a = np.array([4, 6, 1], dtype=float)
b = np.array([3, 4, 5], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.7965
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Sigmoid for multi-class where softmax was required.
- Comparing float results with == instead of np.isclose in tests.

Skills gained choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 104 Sigmoid of 2.75

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of 2.75 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 103, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 105 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

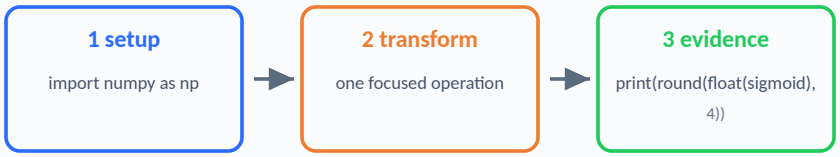
Skill step: moves you from recognition toward independent tracing. It builds on Project 103: cosine similarity. Project 105 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of 2.75 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of 2.75: connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

Reader to Writer Bridge

1. Read: explain sigmoid of 2.75 in one sentence before touching the 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of 2.75 and name what must stay the 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 sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = 2.75
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.9399
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Mixing natural log and log2 between formula and code.
- Dividing by n vs n−1 in variance and wondering about the gap.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 105 Softmax of a 3-Vector (set 27)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 104, where you practised sigmoid of 2.75, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 106 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

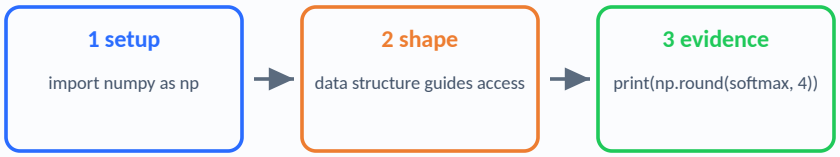
Skill step: moves you from recognition toward independent tracing. It builds on Project 104: sigmoid of 2.75. Project 106 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([-1.0, 1.5, 4.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([-1.0, 1.5, 4.0])`

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. 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	Softmax of a 3-Vector (set 27): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([-1.0, 1.5, 4.0]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([-1.0, 1.5, 4.0])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0062 0.0754 0.9184]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Cosine similarity without normalising — it silently becomes a dot product.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 106 Dot Product (set 28)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 105, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 107 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 105: softmax of a 3-vector. Project 107 changes the surface problem while preserving the underlying method. *progress check*

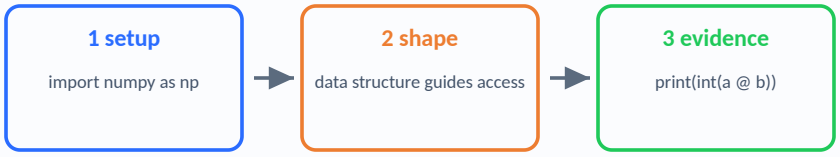
Reading target: dot product is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.

- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([1, 4, 7])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([1, 4, 7])`

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. 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	Dot Product (set 28): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([1, 4, 7]) | Evidence: print(int(a @ b))

🛠️ Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([1, 4, 7])
b = np.array([1, 6, 2])
print(int(a @ b))
```

```
output
39
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.
- Sigmoid for multi-class where softmax was required.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 107 Binary Cross-Entropy (set 29)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 106, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 108 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

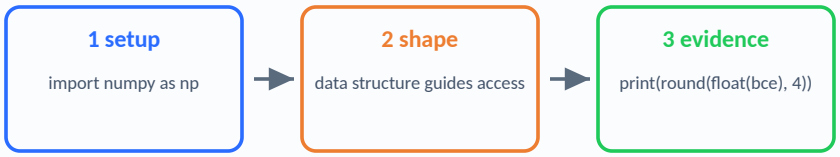
Skill step: moves you from recognition toward independent tracing. It builds on Project 106: dot product. Project 108 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`y = np.array([0, 1, 0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `y = np.array([0, 1, 0])`

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. 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	Binary Cross-Entropy (set 29): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([0, 1, 0]) | Evidence: print(round(float(bce), 4))

Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([0, 1, 0])
p = np.array([0.1, 0.4, 0.7])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.7419
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).
- Mixing natural log and log2 between formula and code.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 108 Cosine Similarity (set 30)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 107, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 109 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

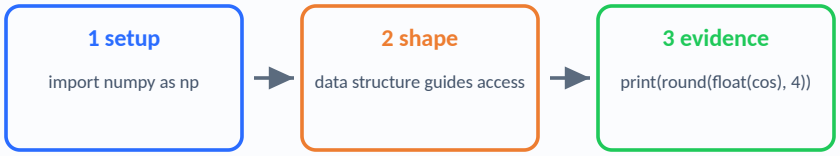
Skill step: pushes the same idea into a less automatic situation. It builds on Project 107: binary cross-entropy. Project 109 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([2, 4, 6], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([2, 4, 6], dtype=float)`

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. 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	Cosine Similarity (set 30): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([2, 4, 6], dtype=float) | Evidence: print(round(float(cos), 4))

🛠️ Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([2, 4, 6], dtype=float)
b = np.array([4, 5, 6], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.9746
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Comparing float results with == instead of np.isclose in tests.
- Cosine similarity without normalising — it silently becomes a dot product.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 109 Binary Cross-Entropy (set 349)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 108, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 110 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

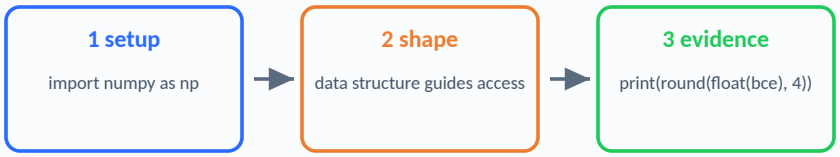
Skill step: pushes the same idea into a less automatic situation. It builds on Project 108: cosine similarity. Project 110 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`y = np.array([0, 1, 0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `y = np.array([0, 1, 0])`

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. 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	Binary Cross-Entropy (set 349): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.

Layer	What to read
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([0, 1, 0]) | Evidence: print(round(float(bce), 4))

Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([0, 1, 0])
p = np.array([0.1, 0.4, 0.7])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.7419
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Dividing by n vs n−1 in variance and wondering about the gap.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 110 Cosine Similarity (set 350)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 109, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 111 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

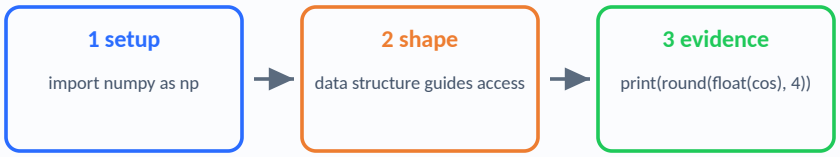
Skill step: pushes the same idea into a less automatic situation. It builds on Project 109: binary cross-entropy. Project 111 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([7, 2, 4], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([7, 2, 4], dtype=float)`

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. 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	Cosine Similarity (set 350): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([7, 2, 4], dtype=float) | Evidence: print(round(float(cos), 4))

Reader to Writer Bridge

1. Read: explain cosine 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 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, Project 111 should feel like a variation, not a brand-new mystery.

from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([7, 2, 4], dtype=float)
b = np.array([5, 6, 7], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.8609
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 111 Softmax of a 3-Vector (set 352)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 110, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 112 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

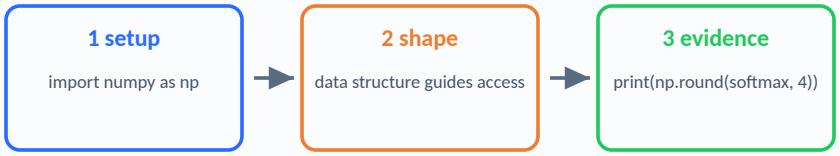
Skill step: pushes the same idea into a less automatic situation. It builds on Project 110: cosine similarity. Project 112 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([0.5, 3.0, -3.5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([0.5, 3.0, -3.5])`

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. 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	Softmax of a 3-Vector (set 352): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([0.5, 3.0, -3.5]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 110.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.
4. Create: 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([0.5, 3.0, -3.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0758 0.9229 0.0014]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Sigmoid for multi-class where softmax was required.
- Comparing float results with == instead of np.isclose in tests.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 112 Sigmoid of -6.0

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -6.0 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 111, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 113 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 111: softmax of a 3-vector. Project 113 changes the surface problem while preserving the underlying method. *progress check*

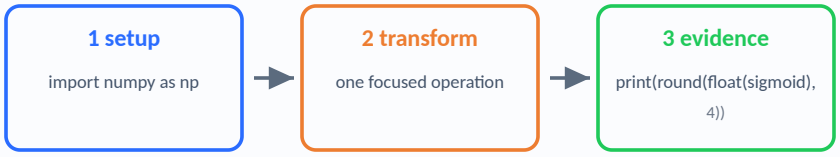
Reading target: sigmoid of -6.0 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.

- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -6.0: connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of -6.0 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 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -6.0 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)

The sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = -6.0
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.0025
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Mixing natural log and log2 between formula and code.
- Dividing by n vs n−1 in variance and wondering about the gap.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 113 Softmax of a 3-Vector (set 2)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 112, where you practised sigmoid of -6.0, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 114 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

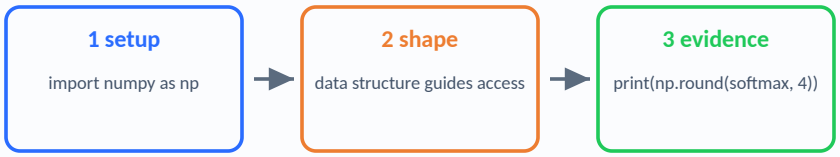
Skill step: pushes the same idea into a less automatic situation. It builds on Project 112: sigmoid of -6.0. Project 114 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([-2.5, 0.0, 2.5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([-2.5, 0.0, 2.5])`

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. 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	Softmax of a 3-Vector (set 2): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([-2.5, 0.0, 2.5]) | Evidence: print(np.round(softmax, 4))

Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([-2.5, 0.0, 2.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0062 0.0754 0.9184]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Cosine similarity without normalising — it silently becomes a dot product.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 114 Dot Product (set 3)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 113, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 115 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

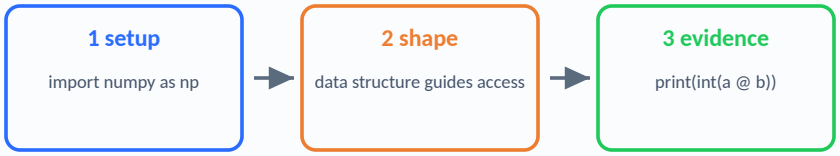
Skill step: pushes the same idea into a less automatic situation. It builds on Project 113: softmax of a 3-vector. Project 115 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([3, 6, 9])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([3, 6, 9])`

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. 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	Dot Product (set 3): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([3, 6, 9]) | Evidence: print(int(a @ b))

🛠️ Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([3, 6, 9])
b = np.array([5, 1, 6])
print(int(a @ b))
```

```
output
75
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.
- Sigmoid for multi-class where softmax was required.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 115 Binary Cross-Entropy (set 4)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 114, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 116 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

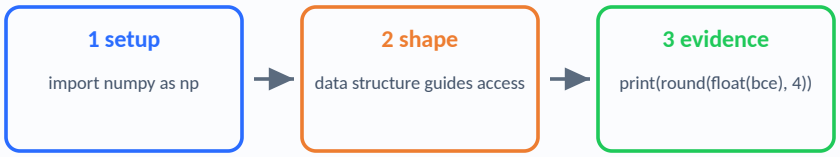
Skill step: pushes the same idea into a less automatic situation. It builds on Project 114: dot product. Project 116 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`y = np.array([1, 0, 1])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `y = np.array([1, 0, 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. 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	Binary Cross-Entropy (set 4): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.

Layer	What to read
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([1, 0, 1]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([1, 0, 1])
p = np.array([0.7, 0.2, 0.5])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).
- Mixing natural log and log2 between formula and code.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 116 Cosine Similarity (set 5)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 115, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 117 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

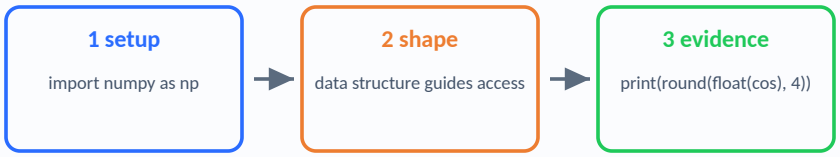
Skill step: pushes the same idea into a less automatic situation. It builds on Project 115: binary cross-entropy. Project 117 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([5, 7, 2], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([5, 7, 2], dtype=float)`

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. 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	Cosine Similarity (set 5): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([5, 7, 2], dtype=float) | Evidence: print(round(float(cos), 4))

Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([5, 7, 2], dtype=float)
b = np.array([6, 7, 1], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.989
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Comparing float results with == instead of np.isclose in tests.
- Cosine similarity without normalising — it silently becomes a dot product.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 117 Sigmoid of -4.25

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -4.25 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 116, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 118 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

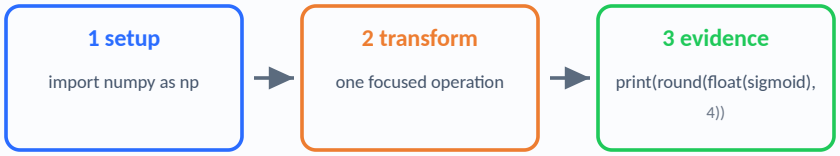
Skill step: pushes the same idea into a less automatic situation. It builds on Project 116: cosine similarity. Project 118 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of -4.25 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -4.25: connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of -4.25 in one sentence before touching the 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -4.25 and name what must stay the 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 sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = -4.25
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.0141
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Dividing by n vs n−1 in variance and wondering about the gap.
- Taking log(0) in cross-entropy — clip probabilities with a tiny epsilon.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 118 Softmax of a 3-Vector (set 7)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 117, where you practised sigmoid of -4.25, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 119 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

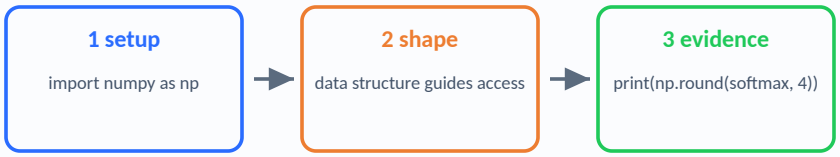
Skill step: pushes the same idea into a less automatic situation. It builds on Project 117: sigmoid of -4.25. Project 119 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`z = np.array([-4.0, -1.5, 1.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([-4.0, -1.5, 1.0])`

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. 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	Softmax of a 3-Vector (set 7): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.

Layer	What to read
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([-4.0, -1.5, 1.0]) | Evidence: print(np.round(softmax, 4))

Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([-4.0, -1.5, 1.0])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0062 0.0754 0.9184]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting that MSE on classification probabilities trains poorly vs cross-entropy.
- Exponentiating big logits raw — overflow; subtract the max first (stable softmax).

Skills gained choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 119 Dot Product (set 8)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Regression & Classification from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 118, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

🧭 Skill Roadmap

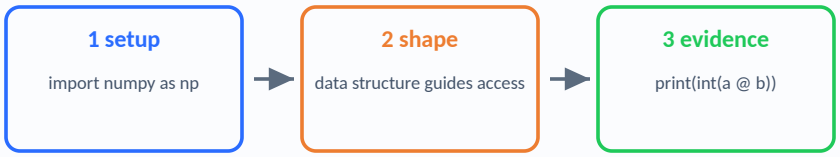
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 118: softmax of a 3-vector. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: linear models, logistic regression and their loss functions, in numpy.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: linear models, logistic regression and their loss functions, in numpy.
2. Apply the rule	`a = np.array([8, 2, 5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Regression & Classification from Scratch becomes practical when you can build the syntax 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 = np.array([8, 2, 5])`

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. 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	Dot Product (set 8): connect the surface task to the chapter skill, linear models, logistic regression and their loss functions, in numpy.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([8, 2, 5]) | Evidence: print(int(a @ b))

Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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, 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([8, 2, 5])
b = np.array([6, 2, 7])
print(int(a @ b))
```

```
output
87
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Sigmoid for multi-class where softmax was required.
- Comparing float results with == instead of np.isclose in tests.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 35 mini-projects, Regression & Classification from Scratch should feel like a reusable Python move: linear models, logistic regression and their loss functions, in numpy.

chapter checkpoint

1 Read
Open any project from Projects 85-119 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 (Optimization & Gradient Descent) 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

Recall · Meaning · Structure · Transfer

7.1.Q1 Recall — Bug hunt: softmax overflows with logits like [1000, 1001]. The maths is right — what is the numerically stable form?

7.1.Q2 Meaning — Bug hunt: cross-entropy returns inf when the model is 100% confident and wrong. Why, and what is the guard?

7.1.Q3 Structure What is the main problem this chapter's topic solves?

- A It gives a named, repeatable way to handle the chapter's programming situation.
- B It removes the need to test code.
- C It makes every program shorter.
- D It only matters for memorising syntax.

7.1.Q4 Transfer What should you identify before running a project?

- A The setup, the rule or transformation, and the evidence line.
- B Only the final answer.
- C Only the imports.
- D Only the file name.

7.1.Q5 Recall What is the most important edge case in Regression & Classification from Scratch?

7.1.Q6 Meaning What proves the skill is transferable?

Answer Key – Chapter 7 – Regression & Classification from Scratch

7.1.Q1. Shift first: exp(z − max(z)) / sum(exp(z − max(z))) — identical result, no overflow.

7.1.Q2. −log(0) is infinite — clip p to [eps, 1−eps] (e.g. 1e-12) before the log.

7.1.Q3. A — Building regression and classification from scratch reveals the loss functions hidden inside libraries.

7.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

7.1.Q5. Probabilities exactly 0 or 1 break log loss, so numerical code often clips values with epsilon.

7.1.Q6. Change w to -2 and explain why the probability flips.

Chapter 8 · Optimization & Gradient Descent

Loss landscapes, learning rates, convergence — the engine of training.

Chapter Skill Map

What this chapter adds: In Optimization & Gradient Descent, the focus is loss landscapes, learning rates, convergence — the engine of training. You practise it across 11 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 8 assumes the reading habits from Chapter 7: Regression & Classification from Scratch. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 9, Neural Networks from Scratch, reuses this chapter's discipline with a new kind of problem.

1 **Projects 120-122**
Recognise the pattern and read the code slowly. *name it first*

2 **Projects 123-125**
Trace similar programs with less prompting and explain what changed.

3 **Projects 126-130**
Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Optimization & Gradient Descent: what problem does this solve for a working programmer?

name the move before typing

Gradient descent is the engine of training: measure loss, compute slope, take a careful step.

This chapter's focus is: Loss landscapes, learning rates, convergence — the engine of training. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
w = 0.0
lr = 0.1
for _ in range(5):
    grad = 2 * (w - 3)
    w -= lr * grad
print(round(w, 3))
```

```
output
2.017
```

Each update moves w 20 percent of the remaining distance toward 3. *predict first, verify second*

LEARNING RATE

The step size used when updating parameters.

LOSS LANDSCAPE

The surface showing loss for different parameter values.

CONVERGENCE

The process of updates approaching a stable good solution.

DIVERGENCE

When updates grow unstable and loss gets worse.

Beginner model

At beginner level, Optimization & Gradient Descent should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Optimisation skill transfers from tiny formulas to neural networks with millions of parameters.

The edge case to respect

Too large a learning rate can overshoot; too small can waste compute.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Try lr=0.01 and lr=0.9, then compare stability and speed.

Example 1

Trace the opening Optimization & Gradient Descent example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read Optimization & Gradient Descent as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Try lr=0.01 and lr=0.9, then compare stability and speed.

⚠ Common Mistakes

- Copying the surface syntax for Optimization & Gradient Descent without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Too large a learning rate can overshoot; too small can waste compute.
- Trusting the output before predicting it by hand.

This chapter contains 11 hands-on mini-projects (Projects 120–130). Type each one out, run it, and experiment.

Project 120 Attention Cost for seq_len=1664

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=1664 into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example. *first anchor* Project 121 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

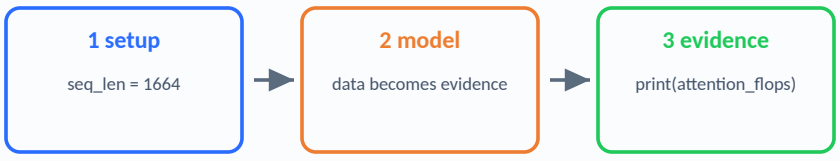
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 121 changes the surface problem while preserving the underlying method. *progress check*

Reading target: attention cost for seq_len=1664 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `seq_len = 1664` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 1664`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=1664: connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. 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: seq_len = 1664 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

Reader to Writer Bridge

1. Read: explain attention cost for seq_len=1664 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 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 1664` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=1664 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 1664
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
2768896
```

How it works:

1. Start at `seq_len = 1664`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Using .item() inside the graph and severing gradients accidentally.
- Shapes silently broadcasting in the loss and averaging the wrong thing.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 121 LLM Cost for 13000 Tokens

💡 The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 13000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 120, where you practised attention cost for seq_len=1664, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 122 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 120: attention cost for seq_len=1664. Project 122 changes the surface problem while preserving the underlying method.

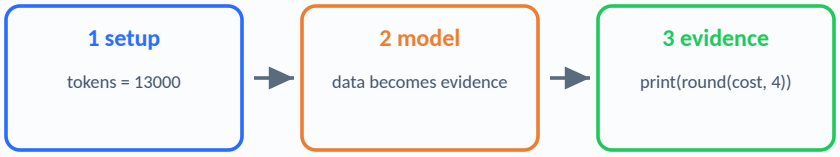
progress check

Reading target: llm cost for 13000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `tokens = 13000` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(round(cost, 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, 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	`tokens = 13000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	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(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Optimization & Gradient Descent becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 13000 Tokens: connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 13000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

🛠️ Reader to Writer Bridge

1. Read: explain llm cost for 13000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 13000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of llm cost for 13000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 13000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.0065
```

How it works:

1. Start at `tokens = 13000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add a second hidden layer and recount parameters.

⚠ Common Mistakes

- Ignoring the quadratic cost of long contexts.
- Mixing tensors across devices (cpu vs mps/cuda).
- Evaluating without model.eval() / torch.no_grad().

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 122 Multi-Head Split (d_model=112)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 121, where you practised llm cost for 13000 tokens, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 123 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 121: llm cost for 13000 tokens. Project 123 changes the surface problem while preserving the underlying method.

progress check

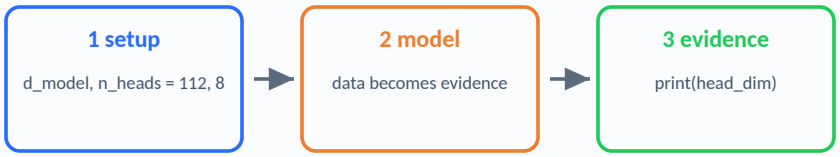
Reading target: multi-head split is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `d_model, n_heads = 112, 8` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 112, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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

Code Reading Stack

Layer	What to read
Intent	Multi-Head Split (d_model=112): connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. 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: d_model, n_heads = 112, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 112, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```python
d_model, n_heads = 112, 8
assert d_model % n_heads == 0, 'must divide evenly'
head_dim = d_model // n_heads
print(head_dim)
```

```
output
14
```

How it works:

1. Start at `d_model, n_heads = 112, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. anchor line
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. evidence line

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Calling .backward() twice on the same graph without retain_graph.
- A learning rate 10× too big — loss oscillates or explodes.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 123 Count Parameters (17->18->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 122, where you practised multi-head split, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 124 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 122: multi-head split. Project 124 changes the surface problem while preserving the underlying method. *progress check*

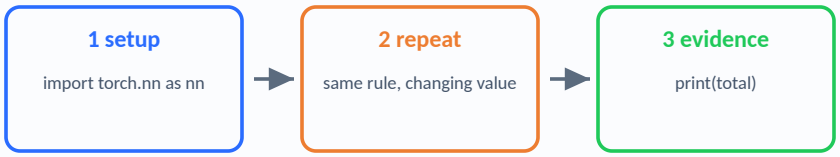
Reading target: count parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import torch.nn as nn` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	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(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Optimization & Gradient Descent becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (17->18->3 MLP): connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: torch, nn, net, Sequential, Linear. 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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

Reader to Writer Bridge

1. Read: explain count 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 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```python
import torch.nn as nn

net = nn.Sequential(nn.Linear(17, 18), nn.ReLU(), nn.Linear(18, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
381
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Add a second hidden layer and recount parameters.

⚠ Common Mistakes

- Ignoring the quadratic cost of long contexts.
- Forgetting optimizer.zero_grad() — gradients accumulate across steps.
- Using .item() inside the graph and severing gradients accidentally.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 124 Attention Cost for seq_len=1792

💡 The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=1792 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 123, where you practised count parameters, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 125 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 123: count parameters. Project 125 changes the surface problem while preserving the underlying method.

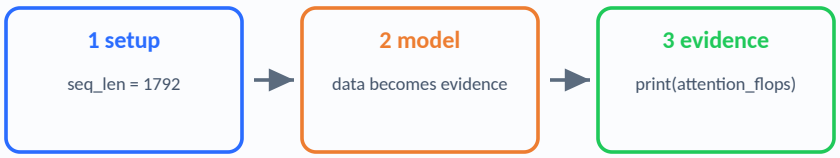
progress check

Reading target: attention cost for seq_len=1792 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `seq_len = 1792` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 1792`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=1792: connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. 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: seq_len = 1792 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

🛠️ Reader to Writer Bridge

1. Read: explain attention cost for seq_len=1792 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 1792` and say why this line, value, import, or entry point is needed before the rest of the Python can work.

3. Transfer: change one small part of attention cost for seq_len=1792 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 1792
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
3211264
```

How it works:

1. Start at `seq_len = 1792`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Comparing tensors with == when allclose was meant.
- Mixing tensors across devices (cpu vs mps/cuda).

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 125 LLM Cost for 14000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 14000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 124, where you practised attention cost for seq_len=1792, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 126 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 124: attention cost for seq_len=1792. Project 126 changes the surface problem while preserving the underlying method.

progress check

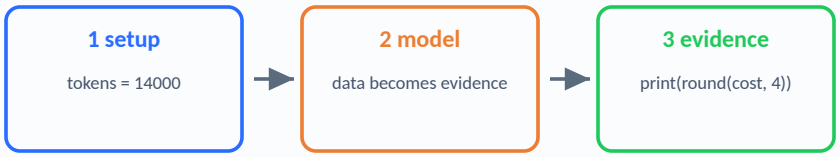
Reading target: llm cost for 14000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `tokens = 14000` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(round(cost, 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, 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	`tokens = 14000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	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(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 14000 Tokens: connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 14000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

🛠️ Reader to Writer Bridge

1. Read: explain llm cost for 14000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 14000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 14000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 14000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.007
```

How it works:

1. Start at `tokens = 14000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add a second hidden layer and recount parameters.

⚠ Common Mistakes

- Ignoring the quadratic cost of long contexts.
- Shapes silently broadcasting in the loss and averaging the wrong thing.
- Calling .backward() twice on the same graph without retain_graph.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 126 Binary Cross-Entropy (set 9)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 125, where you practised llm cost for 14000 tokens, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 127 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

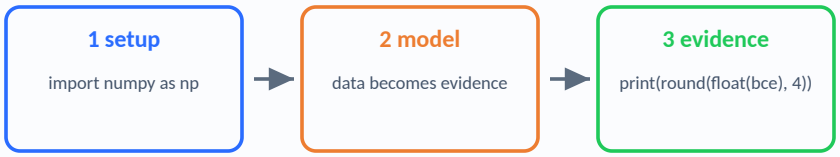
Skill step: moves you from recognition toward independent tracing. It builds on Project 125: llm cost for 14000 tokens. Project 127 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(round(float(bce), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`y = np.array([0, 1, 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(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `y = np.array([0, 1, 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. 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

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	Binary Cross-Entropy (set 9): connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.

Layer	What to read
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([0, 1, 0]) | Evidence: print(round(float(bce), 4))

Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([0, 1, 0])
p = np.array([0.1, 0.4, 0.7])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.7419
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Evaluating without model.eval() / torch.no_grad().
- Forgetting optimizer.zero_grad() — gradients accumulate across steps.

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 127 Cosine Similarity (set 10)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 126, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 128 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

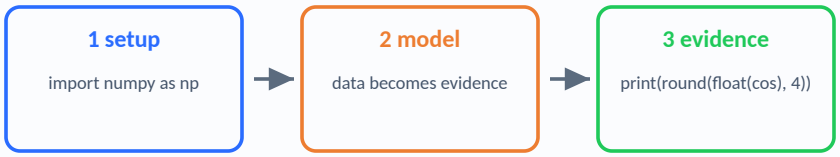
Skill step: pushes the same idea into a less automatic situation. It builds on Project 126: binary cross-entropy. Project 128 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(round(float(cos), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`a = np.array([3, 5, 7], dtype=float)`	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(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Optimization & Gradient Descent becomes practical when you can build the syntax 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 = np.array([3, 5, 7], dtype=float)`

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. 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	Cosine Similarity (set 10): connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([3, 5, 7], dtype=float) | Evidence: print(round(float(cos), 4))

Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([3, 5, 7], dtype=float)
b = np.array([7, 1, 2], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.5975
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- A learning rate 10× too big — loss oscillates or explodes.
- Comparing tensors with == when allclose was meant.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 128 Sigmoid of -2.5

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -2.5 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 127, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 129 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

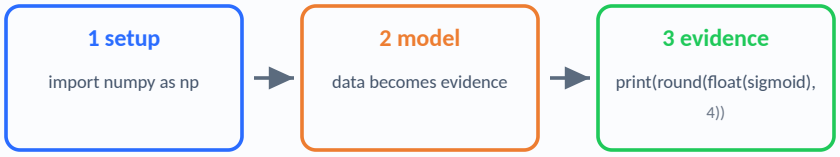
Skill step: pushes the same idea into a less automatic situation. It builds on Project 127: cosine similarity. Project 129 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of -2.5 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(round(float(sigmoid), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	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(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Optimization & Gradient Descent becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -2.5: connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of -2.5 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 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -2.5 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)

The sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = -2.5
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.0759
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Using .item() inside the graph and severing gradients accidentally.
- Shapes silently broadcasting in the loss and averaging the wrong thing.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 129 Softmax of a 3-Vector (set 12)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 128, where you practised sigmoid of -2.5, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 130 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 128: sigmoid of -2.5. Project 130 changes the surface problem while preserving the underlying method. *progress check*

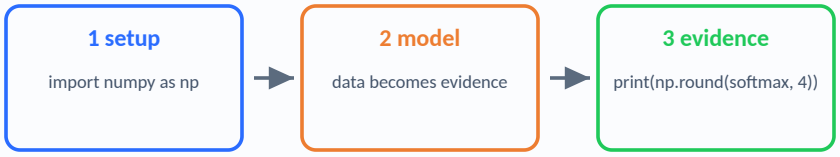
Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` 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: loss landscapes, learning rates, convergence — the engine of training.

- Evidence: `print(np.round(softmax, 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`z = np.array([3.5, -3.0, -0.5])`	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(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `z = np.array([3.5, -3.0, -0.5])`

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. 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	Softmax of a 3-Vector (set 12): connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([3.5, -3.0, -0.5]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([3.5, -3.0, -0.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
```

```
[0.9806 0.0015 0.018 ]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Mixing tensors across devices (cpu vs mps/cuda).
- Evaluating without model.eval() / torch.no_grad().

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 130 Dot Product (set 13)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Optimization & Gradient Descent, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 129, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

🧭 Skill Roadmap

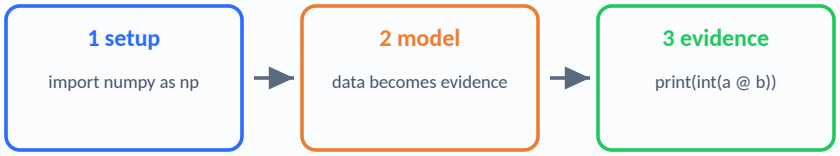
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 129: softmax of a 3-vector. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` 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: loss landscapes, learning rates, convergence — the engine of training.
- Evidence: `print(int(a @ 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: loss landscapes, learning rates, convergence — the engine of training.
2. Apply the rule	`a = np.array([4, 7, 1])`	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(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Optimization & Gradient Descent becomes practical when you can build the syntax 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 = np.array([4, 7, 1])`

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. 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	Dot Product (set 13): connect the surface task to the chapter skill, loss landscapes, learning rates, convergence — the engine of training.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([4, 7, 1]) | Evidence: print(int(a @ b))

Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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, 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([4, 7, 1])
b = np.array([7, 3, 8])
print(int(a @ b))
```

```
output
57
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Calling .backward() twice on the same graph without retain_graph.
- A learning rate 10× too big — loss oscillates or explodes.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 11 mini-projects, Optimization & Gradient Descent should feel like a reusable Python move: loss landscapes, learning rates, convergence — the engine of training. *chapter checkpoint*

1 Read
Open any project from Projects 120-130 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 (Neural Networks from Scratch) 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

Recall · Meaning · Structure · Transfer

8.1.Q1 Recall — Turn back to Project 120 (Attention Cost for seq_len=1664). Without running it, what exactly does it print?

8.1.Q2 Meaning — Turn back to Project 121 (LLM Cost for 13000 Tokens). Without running it, what exactly does it print?

8.1.Q3 Structure Turn back to Project 122 (Multi-Head Split (d_model=112)). Without running it, what exactly does it print?

8.1.Q4 Transfer Bug hunt: loss falls for one step then climbs forever; gradients keep growing. Which single missing line explains it?

8.1.Q5 Recall Bug hunt: `w = torch.tensor([1.0]); y = w * 3; y.backward(); print(w.grad)` prints None. Why?

8.1.Q6 Meaning What is the main problem this chapter's topic solves?

- A It gives a named, repeatable way to handle the chapter's programming situation.
- B It removes the need to test code.
- C It makes every program shorter.
- D It only matters for memorising syntax.

8.1.Q7 Structure What should you identify before running a project?

- A The setup, the rule or transformation, and the evidence line.
- B Only the final answer.
- C Only the imports.
- D Only the file name.

8.1.Q8 Transfer What is the most important edge case in Optimization & Gradient Descent?

8.1.Q9 Recall What proves the skill is transferable?

Answer Key – Chapter 8 – Optimization & Gradient Descent

8.1.Q1. 2768896

8.1.Q2. `0.0065`

8.1.Q3. 14

8.1.Q4. optimizer.zero_grad() — without it each backward() adds onto stale gradients.

8.1.Q5. w was created without requires_grad=True, so autograd never tracked it.

8.1.Q6. A — Gradient descent is the engine of training: measure loss, compute slope, take a careful step.

8.1.Q7. A — prediction comes from setup, rule and evidence, not from guessing.

8.1.Q8. Too large a learning rate can overshoot; too small can waste compute.

8.1.Q9. Try lr=0.01 and lr=0.9, then compare stability and speed.

Chapter 9 · Neural Networks from Scratch

Layers, activations, backprop — a working net in pure NumPy.

Chapter Skill Map

What this chapter adds: In Neural Networks from Scratch, the focus is layers, activations, backprop — a working net in pure numpy. You practise it across 5 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 9 assumes the reading habits from Chapter 8: Optimization & Gradient Descent. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 10, PyTorch Fundamentals, reuses this chapter's discipline with a new kind of problem.

1 **Project 131** Recognise the pattern and read the code slowly. *name it first*	2 **Project 132** Trace similar programs with less prompting and explain what changed.	3 **Projects 133-135** Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Neural Networks from Scratch: what problem does this solve for a working programmer?

name the move before typing

A neural network is stacked linear algebra plus nonlinear activation, trained by backpropagation.

This chapter's focus is: Layers, activations, backprop — a working net in pure NumPy. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```python
import numpy as np

x = np.array([1.0, 2.0])
W = np.array([[1.0, -1.0], [0.5, 1.0]])
z = W @ x
print(np.maximum(z, 0.0))
```

```
output
[0.  2.5]
```

The first raw activation is negative, so ReLU clips it to 0. *predict first, verify second*

LAYER

A transformation from input activations to output activations.

ACTIVATION

A nonlinear function such as ReLU that lets networks model curved relationships.

BACKPROPAGATION

Efficient gradient computation through a chain of operations.

WEIGHT MATRIX

The learned linear transformation inside a layer.

Beginner model

At beginner level, Neural Networks from Scratch should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Deep learning debug work starts with shapes, activations, gradients and loss curves.

The edge case to respect

Without nonlinear activations, many layers collapse into one linear transformation.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Add a bias vector and recompute the activations.

Example 1

Trace the opening Neural Networks from Scratch example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read Neural Networks from Scratch as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Add a bias vector and recompute the activations.

⚠ Common Mistakes

- Copying the surface syntax for Neural Networks from Scratch without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Without nonlinear activations, many layers collapse into one linear transformation.
- Trusting the output before predicting it by hand.

This chapter contains 5 hands-on mini-projects (Projects 131–135). Type each one out, run it, and experiment.

Project 131 Binary Cross-Entropy (set 14)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Neural Networks from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 132 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

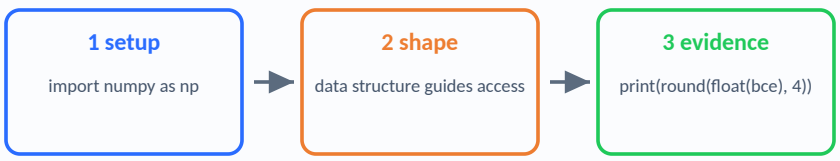
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 132 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: layers, activations, backprop — a working net in pure numpy.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: layers, activations, backprop — a working net in pure numpy.
2. Apply the rule	`y = np.array([1, 0, 1])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `y = np.array([1, 0, 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. 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	Binary Cross-Entropy (set 14): connect the surface task to the chapter skill, layers, activations, backprop — a working net in pure numpy.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([1, 0, 1]) | Evidence: print(round(float(bce), 4))

Reader to Writer Bridge

1. Read: explain binary cross-entropy 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 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([1, 0, 1])
p = np.array([0.3, 0.6, 0.1])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
1.4743
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Counting words when the limit is tokens.
- Comparing embeddings from different models in one space.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 132 Cosine Similarity (set 15)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Neural Networks from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 131, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 133 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

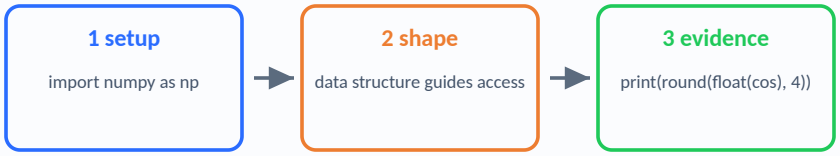
Skill step: moves you from recognition toward independent tracing. It builds on Project 131: binary cross-entropy. Project 133 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: layers, activations, backprop — a working net in pure numpy.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: layers, activations, backprop — a working net in pure numpy.
2. Apply the rule	`a = np.array([1, 3, 5], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Neural Networks from Scratch becomes practical when you can build the syntax 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 = np.array([1, 3, 5], dtype=float)`

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. 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	Cosine Similarity (set 15): connect the surface task to the chapter skill, layers, activations, backprop — a working net in pure numpy.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([1, 3, 5], dtype=float) | Evidence: print(round(float(cos), 4))

🛠️ Reader to Writer Bridge

1. Read: explain cosine 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 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 `print(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([1, 3, 5], dtype=float)
b = np.array([1, 2, 3], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.9939
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting that generation is sampling — set temperature/seed for repeatability.
- Feeding text without the model's own tokenizer.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 133 Sigmoid of -0.75

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -0.75 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Neural Networks from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 132, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 134 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

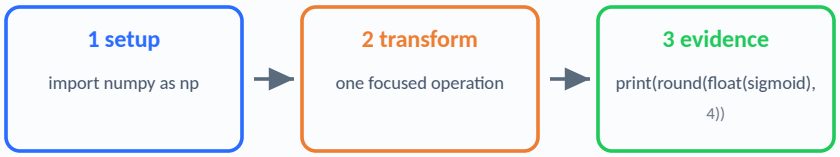
Skill step: moves you from recognition toward independent tracing. It builds on Project 132: cosine similarity. Project 134 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of -0.75 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: layers, activations, backprop — a working net in pure numpy.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: layers, activations, backprop — a working net in pure numpy.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -0.75: connect the surface task to the chapter skill, layers, activations, backprop — a working net in pure numpy.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of -0.75 in one sentence before touching the 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -0.75 and name what must stay the 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 sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = -0.75
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.3208
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Reading attention weights as explanations rather than weights.
- Assuming one token per word — subwords break the estimate.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 134 Softmax of a 3-Vector (set 17)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Neural Networks from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.

- It builds on Project 133, where you practised sigmoid of -0.75, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 135 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

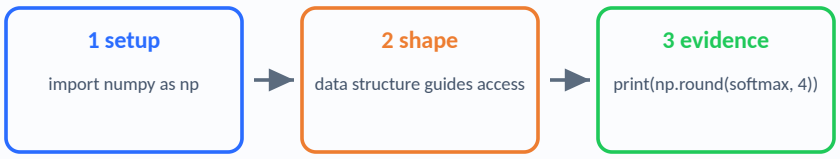
Skill step: pushes the same idea into a less automatic situation. It builds on Project 133: sigmoid of -0.75. Project 135 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: layers, activations, backprop — a working net in pure numpy.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: layers, activations, backprop — a working net in pure numpy.
2. Apply the rule	`z = np.array([2.0, 4.5, -2.0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `z = np.array([2.0, 4.5, -2.0])`

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. 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	Softmax of a 3-Vector (set 17): connect the surface task to the chapter skill, layers, activations, backprop — a working net in pure numpy.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([2.0, 4.5, -2.0]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([2.0, 4.5, -2.0])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0758 0.9229 0.0014]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Truncation silently cutting the question off the end of the prompt.
- Counting words when the limit is tokens.

Skills gained choosing and reading data structures | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 135 Dot Product (set 18)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Neural Networks from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 134, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

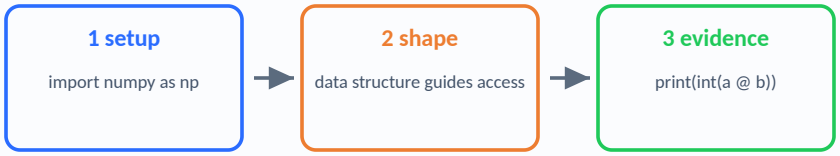
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 134: softmax of a 3-vector. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: layers, activations, backprop — a working net in pure numpy.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: layers, activations, backprop — a working net in pure numpy.
2. Apply the rule	`a = np.array([9, 3, 6])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Neural Networks from Scratch becomes practical when you can build the syntax 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 = np.array([9, 3, 6])`

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. 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	Dot Product (set 18): connect the surface task to the chapter skill, layers, activations, backprop — a working net in pure numpy.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([9, 3, 6]) | Evidence: print(int(a @ b))

🛠️ Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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, 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([9, 3, 6])
b = np.array([8, 4, 9])
print(int(a @ b))
```

```
output
138
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Ignoring special tokens ([CLS], <s>) in length budgets.
- Forgetting that generation is sampling — set temperature/seed for repeatability.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 5 mini-projects, Neural Networks from Scratch should feel like a reusable Python move: layers, activations, backprop — a working net in pure numpy. *chapter checkpoint*

1 Read
Open any project from Projects 131-135 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 (PyTorch Fundamentals) 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

Recall · Meaning · Structure · Transfer

9.1.Q1 Meaning — Bug hunt: a prompt "fits" at 4,000 words but the API rejects it at 8k context. What was measured wrong?

9.1.Q2 Meaning — Bug hunt: cosine similarity between sentence vectors from model A and model B is meaningless. Why?

9.1.Q3 Structure — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

9.1.Q4 Transfer — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

9.1.Q5 Recall — What is the most important edge case in Neural Networks from Scratch?

9.1.Q6 Meaning — What proves the skill is transferable?

Answer Key – Chapter 9 – Neural Networks from Scratch

9.1.Q1. Words ≠ tokens — 4,000 English words is roughly 5–6k tokens; budget with the tokenizer, not wc.

9.1.Q2. Different models define different embedding spaces — vectors are only comparable within one model’ s space.

9.1.Q3. A — A neural network is stacked linear algebra plus nonlinear activation, trained by backpropagation.

9.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

9.1.Q5. Without nonlinear activations, many layers collapse into one linear transformation.

9.1.Q6. Add a bias vector and recompute the activations.

Chapter 10 · PyTorch Fundamentals

Tensors, autograd, modules and training loops.

Chapter Skill Map

What this chapter adds: In PyTorch Fundamentals, the focus is tensors, autograd, modules and training loops. You practise it across 10 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 10 assumes the reading habits from Chapter 9: Neural Networks from Scratch. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 11, Vision: Images as Arrays, reuses this chapter's discipline with a new kind of problem.

1 **Projects 136-138**
Recognise the pattern and read the code slowly. *name it first*

2 **Projects 139-141**
Trace similar programs with less prompting and explain what changed.

3 **Projects 142-145**
Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

PyTorch Fundamentals: what problem does this solve for a working programmer?

name the move before typing

PyTorch adds tensors, autograd and modules so training loops can be written directly and inspected.

This chapter's focus is: Tensors, autograd, modules and training loops. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
import torch

x = torch.tensor([1.0, 2.0], requires_grad=True)
y = (x * x).sum()
y.backward()
print(x.grad)
```

```
output
tensor([2., 4.])
```

The derivative of x squared is 2x, so the gradient is [2, 4]. *predict first, verify second*

TENSOR

A multidimensional array used by PyTorch.

AUTOGRAD

PyTorch's automatic differentiation system.

MODULE

A reusable neural-network component with parameters.

TRAINING LOOP

The repeated sequence forward pass, loss, backward pass, optimizer step.

Beginner model

At beginner level, PyTorch Fundamentals should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

PyTorch is powerful because it keeps the math visible while handling differentiation.

The edge case to respect

Forgetting zero_grad accumulates gradients across steps and changes training behavior.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Create a scalar parameter w and compute the gradient of (w - 5)^2.

Example 1

Trace the opening PyTorch Fundamentals example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read PyTorch Fundamentals as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Create a scalar parameter w and compute the gradient of (w - 5)^2.

⚠ Common Mistakes

- Copying the surface syntax for PyTorch Fundamentals without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Forgetting zero_grad accumulates gradients across steps and changes training behavior.
- Trusting the output before predicting it by hand.

This chapter contains 10 hands-on mini-projects (Projects 136–145). Type each one out, run it, and experiment.

Project 136 Autograd: Compute a Gradient

💡 The Idea Behind It

PyTorch is the dominant deep-learning framework: tensors with autograd, composable layers and optimisers. The deeper point is learning how Python turns a small intention like autograd: compute a gradient into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 137 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

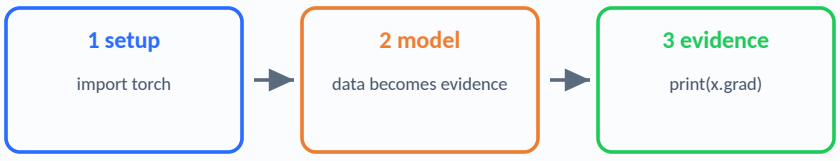
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 137 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import torch` 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: tensors, autograd, modules and training loops.
- Evidence: `print(x.grad)` exposes the result; match it 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	`import torch`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`y = (x ** 2).sum()`	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(x.grad)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `y = (x ** 2).sum()`

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. 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	Autograd: Compute a Gradient: connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: torch, tensor, requires_grad, sum, backward. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import torch | Core: y = (x ** 2).sum() | Evidence: print(x.grad)

Reader to Writer Bridge

1. Read: explain autograd: compute a gradient 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 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(x.grad)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of autograd: compute a gradient and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

backward() fills .grad with d(y)/d(x); here that is 2x.

```
import torch

x = torch.tensor([1., 2., 3.], requires_grad=True)
y = (x ** 2).sum()
y.backward()
print(x.grad)
```

```
output
tensor([2., 4., 6.])
```

How it works:

1. Start at `import torch`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(x.grad)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Add another layer.

⚠ Common Mistakes

- Forgetting opt.zero_grad() — gradients accumulate.
- Stuffing the context window full and starving the answer of room.
- Embedding the query with a different model than the documents.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 137 Define a Feed-Forward Network

💡 The Idea Behind It

PyTorch is the dominant deep-learning framework: tensors with autograd, composable layers and optimisers. The deeper point is learning how Python turns a small intention like define a feed-forward network into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 136, where you practised autograd: compute a gradient, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 138 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 136: autograd: compute a gradient. Project 138 changes the surface problem while preserving the underlying method.

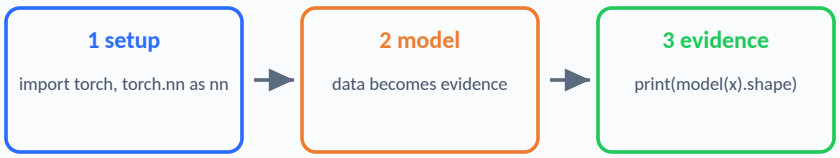
progress check

Reading target: define a feed-forward network is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import torch, torch.nn as nn` 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: tensors, autograd, modules and training loops.
- Evidence: `print(model(x).shape)` exposes the result; match it 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	`import torch, torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`model = nn.Sequential(`	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(model(x).shape)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `model = nn.Sequential(`

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. 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	Define a Feed-Forward Network: connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: torch, nn, model, Sequential, Linear. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import torch, torch.nn as nn | Core: model = nn.Sequential(| Evidence: print(model(x).shape)

🛠️ Reader to Writer Bridge

1. Read: explain define a feed-forward network in one sentence before touching the 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(model(x).shape)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch, torch.nn as nn` and say why 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 a feed-forward network and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

nn.Sequential stacks layers; the output has 3 features.

```
import torch, torch.nn as nn

model = nn.Sequential(
    nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 3))
x = torch.randn(1, 4)
print(model(x).shape)
```

```
output
torch.Size([1, 3])
```

How it works:

1. Start at `import torch, torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(model(x).shape)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Switch SGD for Adam and compare.

⚠ Common Mistakes

- Shape mismatches between layers.
- No overlap between chunks, cutting answers in half at boundaries.
- Not citing sources, so wrong answers cannot be traced.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 138 Cross-Entropy Loss

The Idea Behind It

PyTorch is the dominant deep-learning framework: tensors with autograd, composable layers and optimisers. The deeper point is learning how Python turns a small intention like cross-entropy loss into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 137, where you practised define a feed-forward network, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 139 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 137: define a feed-forward network. Project 139 changes the surface problem while preserving the underlying method. *progress check*

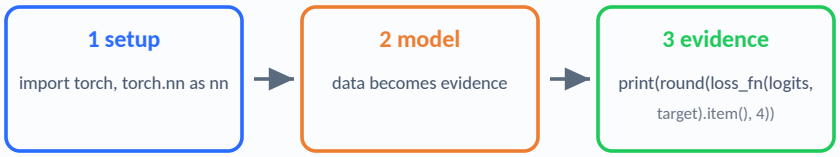
Reading target: cross-entropy loss is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import torch, torch.nn as nn` 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: tensors, autograd, modules and training loops.
- Evidence: `print(round(loss_fn(logits, target).item(), 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, 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	`import torch, torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`loss_fn = nn.CrossEntropyLoss()`	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(round(loss_fn(logits, target).item(), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `loss_fn = nn.CrossEntropyLoss()`

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. 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	Cross-Entropy Loss: connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: torch, nn, loss_fn, CrossEntropyLoss, logits. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import torch, torch.nn as nn | Core: loss_fn = nn.CrossEntropyLoss() | Evidence: print(round(loss_fn(logits, target).item(), 4))

Reader to Writer Bridge

1. Read: explain cross-entropy loss in one sentence before touching the 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(round(loss_fn(logits, target).item(), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch, torch.nn as nn` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of cross-entropy loss and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

CrossEntropyLoss combines softmax and negative log-likelihood.

```python
import torch, torch.nn as nn

loss_fn = nn.CrossEntropyLoss()
logits = torch.tensor([[2.0, 0.5, 0.1]])
target = torch.tensor([0])
print(round(loss_fn(logits, target).item(), 4))
```

```
output
0.3168
```

How it works:

1. Start at `import torch, torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(loss_fn(logits, target).item(), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output.

evidence line

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another layer.

⚠ Common Mistakes

- Forgetting opt.zero_grad() — gradients accumulate.
- Chunks so small they lose the sentence that answers the question.
- Retrieving top-k by raw dot product on unnormalised embeddings.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 139 A Gradient-Descent Loop

💡 The Idea Behind It

PyTorch is the dominant deep-learning framework: tensors with autograd, composable layers and optimisers. The deeper point is learning how Python turns a small intention like a gradient-descent loop into a readable

sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 138, where you practised cross-entropy loss, and asks you to reuse the same reasoning pattern. reuse the pattern Project 140 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

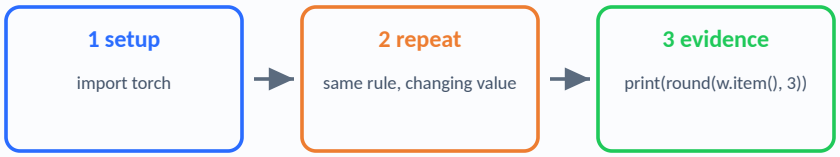
Skill step: moves you from recognition toward independent tracing. It builds on Project 138: cross-entropy loss. Project 140 changes the surface problem while preserving the underlying method. progress check

Reading target: a gradient-descent loop is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import torch` 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: tensors, autograd, modules and training loops.
- Evidence: `print(round(w.item(), 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 torch`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`for _ in range(opt_steps):`	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(round(w.item(), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: PyTorch Fundamentals becomes practical when you can build the syntax 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 _ in range(opt_steps):`

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. 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

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	A Gradient-Descent Loop: connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: torch, opt_steps, tensor, requires_grad, opt. 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 torch | Core: for _ in range(opt_steps): | Evidence: print(round(w.item(), 3))

🛠️ Reader to Writer Bridge

1. Read: explain a gradient-descent 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 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(round(w.item(), 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of a gradient-descent 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)

Each step nudges w toward the value that minimises the loss.

```
import torch

opt_steps = 3
w = torch.tensor([5.0], requires_grad=True)
opt = torch.optim.SGD([w], lr=0.1)
for _ in range(opt_steps):
    loss = (w - 1) ** 2
    opt.zero_grad(); loss.backward(); opt.step()
print(round(w.item(), 3))
```

```
output
3.048
```

How it works:

1. Start at `import torch`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(w.item(), 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Switch SGD for Adam and compare.

⚠ Common Mistakes

- Shape mismatches between layers.
- Chunks so large that retrieval returns one blob, so the model quotes filler.
- Stuffing the context window full and starving the answer of room.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 140 A Convolution Layer

💡 The Idea Behind It

PyTorch is the dominant deep-learning framework: tensors with autograd, composable layers and optimisers. The deeper point is learning how Python turns a small intention like a convolution layer into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 139, where you practised a gradient-descent loop, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 141 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 139: a gradient-descent loop. Project 141 changes the surface problem while preserving the underlying method.

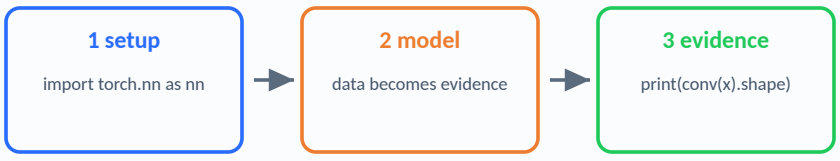
progress check

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

🔎 Logic Lens

- Setup: `import torch.nn as nn` 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: tensors, autograd, modules and training loops.
- Evidence: `print(conv(x).shape)` exposes the result; match it 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	`import torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3)`	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(conv(x).shape)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3)`

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. 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	A Convolution Layer: connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: torch, nn, conv, Conv2d, in_channels. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: import torch.nn as nn | Core: conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3) | Evidence: print(conv(x).shape)

Reader to Writer Bridge

1. Read: explain a convolution layer in one sentence before touching the 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(conv(x).shape)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of a convolution layer and name what must stay the 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 3x3 conv shrinks 28x28 to 26x26 and yields 4 feature maps.

```
import torch.nn as nn

conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3)
import torch
x = torch.randn(1, 1, 28, 28)
print(conv(x).shape)
```

```
output
torch.Size([1, 4, 26, 26])
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(conv(x).shape)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Add another layer.

⚠ Common Mistakes

- Forgetting opt.zero_grad() — gradients accumulate.
- Caching nothing and re-embedding the corpus every run.
- No overlap between chunks, cutting answers in half at boundaries.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 141 Column Sums of a 7x7 Matrix (set 65)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like column sums of a 7x7 matrix into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 140, where you practised a convolution layer, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 142 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

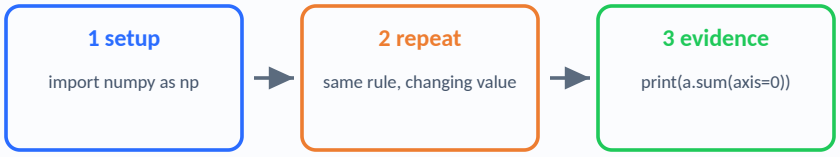
Skill step: moves you from recognition toward independent tracing. It builds on Project 140: a convolution layer. Project 142 changes the surface problem while preserving the underlying method. *progress check*

Reading target: column sums of a 7x7 matrix is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` 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: tensors, autograd, modules and training loops.
- Evidence: `print(a.sum(axis=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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`a = np.arange(1, 50).reshape(7, 7)`	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(a.sum(axis=0))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: PyTorch Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 50).reshape(7, 7)`

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. 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	Column Sums of a 7x7 Matrix (set 65): connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: numpy, np, arange, reshape, sum. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 50).reshape(7, 7) | Evidence: print(a.sum(axis=0))

🛠️ Reader to Writer Bridge

1. Read: explain column sums of a 7x7 matrix in one sentence before touching the 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(a.sum(axis=0))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of column sums of a 7x7 matrix and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

axis=0 sums down each column.

```
import numpy as np

a = np.arange(1, 50).reshape(7, 7)
print(a.sum(axis=0))
```

```
output
[154 161 168 175 182 189 196]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a.sum(axis=0))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Embedding the query with a different model than the documents.
- Chunks so small they lose the sentence that answers the question.

Skills gained tracing repeated work | choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 142 Summarise arange 1..17

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like summarise arange 1..17 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 141, where you practised column sums of a 7x7 matrix, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 143 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

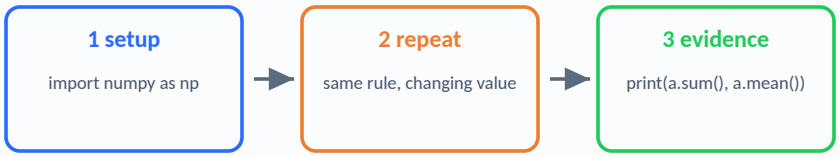
Skill step: pushes the same idea into a less automatic situation. It builds on Project 141: column sums of a 7x7 matrix. Project 143 changes the surface problem while preserving the underlying method. *progress check*

Reading target: summarise arange 1..17 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` 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: tensors, autograd, modules and training loops.
- Evidence: `print(a.sum(), a.mean())` exposes the result; match it against the output before changing the program. *output proves the trace*

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`a = np.arange(1, 18)`	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(a.sum(), a.mean())`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: PyTorch Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 18)`

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. 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	Summarise arange 1..17: connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: numpy, np, arange, sum, mean. Names are the reader's map of roles and responsibilities.

Layer	What to read
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 18) | Evidence: print(a.sum(), a.mean())

🛠️ Reader to Writer Bridge

1. Read: explain summarise arange 1..17 in one sentence before touching the 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(a.sum(), a.mean())`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 arange 1..17 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.arange builds a numeric range; sum/mean reduce it.

```python
import numpy as np

a = np.arange(1, 18)
print(a.sum(), a.mean())
```

```
output
153 9.0
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a.sum(), a.mean())`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Not citing sources, so wrong answers cannot be traced.
- Chunks so large that retrieval returns one blob, so the model quotes filler.

Skills gained tracing repeated work | choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 143 Sort an Array (set 67)

The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like sort an array into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*

- This project pushes the same idea into a less automatic situation.
- It builds on Project 142, where you practised summarise arange 1..17, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 144 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

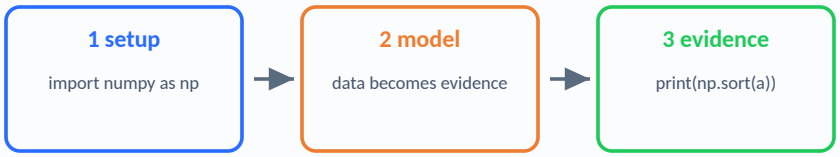
Skill step: pushes the same idea into a less automatic situation. It builds on Project 142: summarise arange 1..17. Project 144 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `import numpy as np` 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: tensors, autograd, modules and training loops.
- Evidence: `print(np.sort(a))` exposes the result; match it 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`a = np.array([13, 24, 35, 46, 57])`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something

Move	Code to inspect	Why it matters
		measurable. slow line
3. Check the evidence	`print(np.sort(a))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: PyTorch Fundamentals becomes practical when you can build the syntax 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 = np.array([13, 24, 35, 46, 57])`

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. 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	Sort an Array (set 67): connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: numpy, np, array, sort. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([13, 24, 35, 46, 57]) | Evidence: print(np.sort(a))

Reader to Writer Bridge

1. Read: explain sort an array in one sentence before touching the 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(np.sort(a))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 an array and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

np.sort returns a new, ascending copy of the array.

```
import numpy as np

a = np.array([13, 24, 35, 46, 57])
print(np.sort(a))
```

```
output
[13 24 35 46 57]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.sort(a))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- Retrieving top-k by raw dot product on unnormalised embeddings.
- Caching nothing and re-embedding the corpus every run.

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 144 Scale arange 1..7 by 6

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like scale arange 1..7 by 6 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 143, where you practised sort an array, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 145 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 143: sort an array. Project 145 changes the surface problem while preserving the underlying method. *progress check*

Reading target: scale arange 1..7 by 6 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

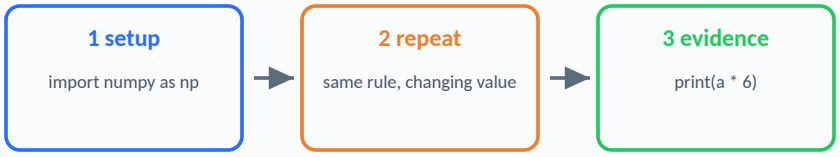
🔎 Logic Lens

- Setup: `import numpy as np` 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: tensors, autograd, modules and training loops.
- Evidence: `print(a * 6)` exposes the result; match it against the output before changing the program.

output proves the trace

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`a = np.arange(1, 8)`	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(a * 6)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: PyTorch Fundamentals becomes practical when you can build the syntax 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 = np.arange(1, 8)`

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. 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	Scale arange 1..7 by 6: connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: numpy, np, arange. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; loops repeat a rule.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.arange(1, 8) | Evidence: print(a * 6)

🛠️ Reader to Writer Bridge

1. Read: explain scale arange 1..7 by 6 in one sentence before touching the 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(a * 6)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scale arange 1..7 by 6 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🤖 From Concept to Code (Python)

Arithmetic on arrays is element-wise — no loop needed.

```
import numpy as np
```

```
a = np.arange(1, 8)
print(a * 6)
```

```
output
[ 6 12 18 24 30 36 42]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(a * 6)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change the array and re-run.

⚠ Common Mistakes

- Looping element-by-element instead of vectorising.
- Stuffing the context window full and starving the answer of room.
- Embedding the query with a different model than the documents.

Skills gained tracing repeated work | choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 145 Boolean Mask: Above Mean (set 69)

💡 The Idea Behind It

NumPy arrays are the foundation of all numerical AI work — fast, vectorised and the input format every model expects. The deeper point is learning how Python turns a small intention like boolean mask: above mean into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, PyTorch Fundamentals, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 144, where you practised scale arange 1..7 by 6, and asks you to reuse the same reasoning pattern. reuse the pattern After this, you should be able to explain the chapter pattern without leaning on the exact values used here. own the pattern

🧭 Skill Roadmap

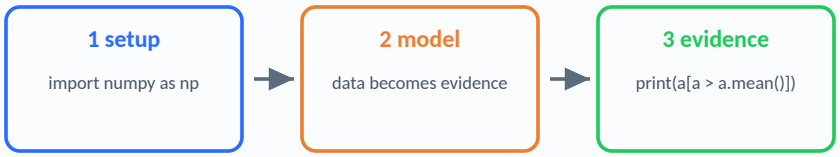
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 144: scale arange 1..7 by 6. By the end of this chapter, you should be able to explain the pattern without copying the exact example. progress check

Reading target: boolean mask: above mean is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import numpy as np` 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: tensors, autograd, modules and training loops.
- Evidence: `print(a[a > a.mean()])` exposes the result; match it 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tensors, autograd, modules and training loops.
2. Apply the rule	`a = np.array([4, 9, 14, 19, 24, 29])`	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(a[a > a.mean()])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: PyTorch Fundamentals becomes practical when you can build the syntax 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 = np.array([4, 9, 14, 19, 24, 29])`

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. 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	Boolean Mask: Above Mean (set 69): connect the surface task to the chapter skill, tensors, autograd, modules and training loops.
Names	Watch these names first: numpy, np, array, mean. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([4, 9, 14, 19, 24, 29]) | Evidence: print(a[a > a.mean()])

🛠️ Reader to Writer Bridge

1. Read: explain boolean mask: above mean in one sentence before touching the 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, 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(a[a > a.mean()])`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of boolean mask: above mean and name what must stay the 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 boolean array selects elements where the test is True.

```python
import numpy as np

a = np.array([4, 9, 14, 19, 24, 29])
print(a[a > a.mean()])
```

```
output
[19 24 29]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(a[a > a.mean()])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Add another reduction (std, median).

⚠ Common Mistakes

- Mixing up axis=0 (columns) and axis=1 (rows).
- No overlap between chunks, cutting answers in half at boundaries.
- Not citing sources, so wrong answers cannot be traced.

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 10 mini-projects, PyTorch Fundamentals should feel like a reusable Python move: tensors, autograd, modules and training loops. *chapter checkpoint*

1 Read
Open any project from Projects 136-145 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 (Vision: Images as Arrays) 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

Recall · Meaning · Structure · Transfer

10.1.Q1 Recall Bug hunt: RAG answers confidently about a topic absent from the corpus. What instruction (and check) reins it in?

10.1.Q2 Meaning Bug hunt: answers degrade after switching the embedding model for queries only. Why?

10.1.Q3 Structure What is the main problem this chapter's topic solves?

- A It gives a named, repeatable way to handle the chapter's programming situation.
- B It removes the need to test code.
- C It makes every program shorter.
- D It only matters for memorising syntax.

10.1.Q4 Transfer What should you identify before running a project?

- A The setup, the rule or transformation, and the evidence line.
- B Only the final answer.
- C Only the imports.
- D Only the file name.

10.1.Q5 Recall What is the most important edge case in PyTorch Fundamentals?

10.1.Q6 Transfer What proves the skill is transferable?

Answer Key – Chapter 10 – PyTorch Fundamentals

10.1.Q1. "Answer only from the provided context; say *not found* otherwise" — plus verifying retrieval scores clear a threshold.

10.1.Q2. Query and document vectors now live in different spaces — re-embed the corpus with the same model.

10.1.Q3. A — PyTorch adds tensors, autograd and modules so training loops can be written directly and inspected.

10.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

10.1.Q5. Forgetting zero_grad accumulates gradients across steps and changes training behavior.

10.1.Q6. Create a scalar parameter w and compute the gradient of (w - 5)^2.



PART III

Advanced Mastery

Push into the professional toolkit: concurrency, design patterns, performance, data work and the algorithms behind real systems.

Chapter 11 – Vision: Images as Arrays · Chapter 12 – NLP & Text Processing · Chapter 13 – Embeddings & Similarity Search · Chapter 14 – Transformers & Attention from Scratch · Chapter 15 – LLM Engineering, Prompts & RAG · Chapter 16 – Scaling, Compute, Cost & MLOps

Chapter 11 · Vision: Images as Arrays

Pixels, channels, convolutions and image features.

Chapter Skill Map

What this chapter adds: In Vision: Images as Arrays, the focus is pixels, channels, convolutions and image features. You practise it across 5 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 11 assumes the reading habits from Chapter 10: PyTorch Fundamentals. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 12, NLP & Text Processing, reuses this chapter's discipline with a new kind of problem.

1 **Project 146** Recognise the pattern and read the code slowly. *name it first*	2 **Project 147** Trace similar programs with less prompting and explain what changed.	3 **Projects 148-150** Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Vision: Images as Arrays: what problem does this solve for a working programmer?

name the move before typing

Images are arrays with height, width and channels; vision begins by respecting that shape.

This chapter's focus is: Pixels, channels, convolutions and image features. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```python
import numpy as np

image = np.zeros((2, 2, 3), dtype=int)
image[0, 0] = [255, 0, 0]
print(image.shape)
print(image[0, 0].tolist())
```

```
output
(2, 2, 3)
[255, 0, 0]
```

The red pixel lives at row 0, column 0, across three channel values. *predict first, verify second*

PIXEL

One location in an image grid.

CHANNEL

A colour or feature dimension such as red, green and blue.

CONVOLUTION

A sliding-window operation that detects local patterns.

FEATURE MAP

The output grid produced by a filter or layer.

Beginner model

At beginner level, Vision: Images as Arrays should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Vision models build from this simple grid idea to filters, features and spatial hierarchies.

The edge case to respect

Channel order matters: RGB and BGR swaps can silently ruin a model input.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Create a grayscale 3x3 image and compute its mean brightness.

Example 1

Trace the opening Vision: Images as Arrays example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read Vision: Images as Arrays as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Create a grayscale 3x3 image and compute its mean brightness.

⚠ Common Mistakes

- Copying the surface syntax for Vision: Images as Arrays without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Channel order matters: RGB and BGR swaps can silently ruin a model input.
- Trusting the output before predicting it by hand.

This chapter contains 5 hands-on mini-projects (Projects 146–150). Type each one out, run it, and experiment.

Project 146 Binary Cross-Entropy (set 19)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Vision: Images as Arrays, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example. *first anchor* Project 147 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

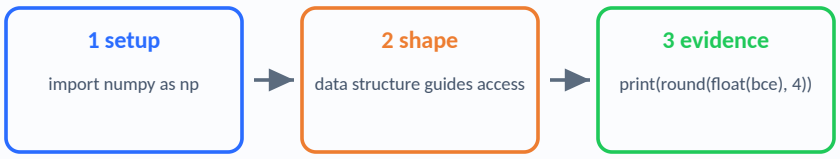
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 147 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: pixels, channels, convolutions and image features.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: pixels, channels, convolutions and image features.
2. Apply the rule	`y = np.array([0, 1, 0])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `y = np.array([0, 1, 0])`

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. 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	Binary Cross-Entropy (set 19): connect the surface task to the chapter skill, pixels, channels, convolutions and image features.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([0, 1, 0]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy 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 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([0, 1, 0])
p = np.array([0.5, 0.8, 0.3])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Quoting GPU memory for weights and forgetting activations/optimizer state.
- Comparing fp16 and int8 numbers as if accuracy were free.

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 147 Cosine Similarity (set 20)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Vision: Images as Arrays, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 146, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 148 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

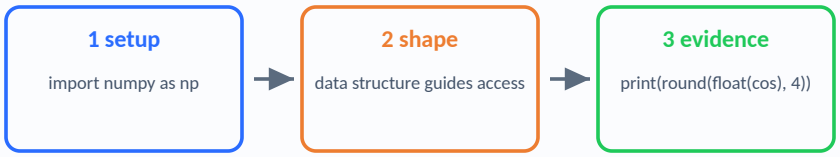
Skill step: moves you from recognition toward independent tracing. It builds on Project 146: binary cross-entropy. Project 148 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: pixels, channels, convolutions and image features.
- Evidence: `print(round(float(cos), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: pixels, channels, convolutions and image features.
2. Apply the rule	`a = np.array([6, 1, 3], dtype=float)`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Vision: Images as Arrays becomes practical when you can build the syntax 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 = np.array([6, 1, 3], dtype=float)`

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. 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	Cosine Similarity (set 20): connect the surface task to the chapter skill, pixels, channels, convolutions and image features.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([6, 1, 3], dtype=float) | Evidence: print(round(float(cos), 4))

🛠️ Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([6, 1, 3], dtype=float)
b = np.array([2, 3, 4], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.7392
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Confusing parameters with tokens in the 6·N·D estimate.
- Pricing only output tokens when the prompt dominates the bill.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 148 Sigmoid of 1.0

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of 1.0 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Vision: Images as Arrays, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 147, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 149 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

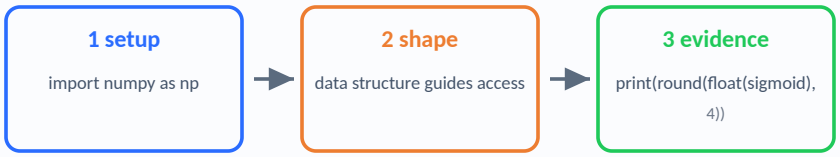
Skill step: moves you from recognition toward independent tracing. It builds on Project 147: cosine similarity. Project 149 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of 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 numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: pixels, channels, convolutions and image features.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: pixels, channels, convolutions and image features.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. slow line
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of 1.0: connect the surface task to the chapter skill, pixels, channels, convolutions and image features.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of 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 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of 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)

The sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = 1.0
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.7311
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Estimating fine-tune cost with full-training formulas (LoRA changes the game).
- Ignoring that Adam roughly triples weight memory during training.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 149 Softmax of a 3-Vector (set 22)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Vision: Images as Arrays, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.

- It builds on Project 148, where you practised sigmoid of 1.0, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 150 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

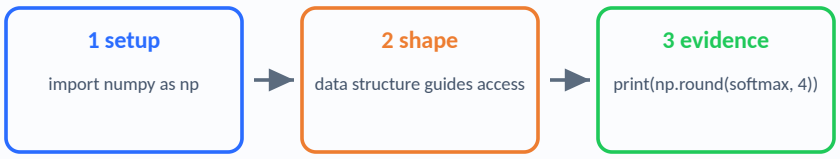
Skill step: pushes the same idea into a less automatic situation. It builds on Project 148: sigmoid of 1.0. Project 150 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: pixels, channels, convolutions and image features.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: pixels, channels, convolutions and image features.
2. Apply the rule	`z = np.array([0.5, 3.0, -3.5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `z = np.array([0.5, 3.0, -3.5])`

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. 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	Softmax of a 3-Vector (set 22): connect the surface task to the chapter skill, pixels, channels, convolutions and image features.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([0.5, 3.0, -3.5]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([0.5, 3.0, -3.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0758 0.9229 0.0014]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Plot the function across a range of values.

⚠ Common Mistakes

- Forgetting an epsilon inside log().
- Mixing GB (10^9) and GiB (2^{30}) mid-calculation.
- Quoting GPU memory for weights and forgetting activations/optimizer state.

Skills gained choosing and reading data structures predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 150 Dot Product (set 23)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Vision: Images as Arrays, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 149, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. reuse the pattern After this, you should be able to explain the chapter pattern without leaning on the exact values used here. own the pattern

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 149: softmax of a 3-vector. By the end of this chapter, you should be able to explain the pattern without copying the exact example. progress check

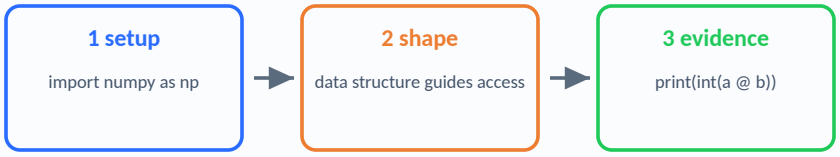
Reading target: dot product is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

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

- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: pixels, channels, convolutions and image features.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: pixels, channels, convolutions and image features.
2. Apply the rule	`a = np.array([5, 8, 2])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Vision: Images as Arrays becomes practical when you can build the syntax 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 = np.array([5, 8, 2])`

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. 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	Dot Product (set 23): connect the surface task to the chapter skill, pixels, channels, convolutions and image features.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([5, 8, 2]) | Evidence: print(int(a @ b))

🛠️ Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([5, 8, 2])
b = np.array([9, 5, 1])
print(int(a @ b))
```

```
output
87
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Assuming batch size 1 latency scales linearly to batch 32.
- Confusing parameters with tokens in the 6·N·D estimate.

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 5 mini-projects, Vision: Images as Arrays should feel like a reusable Python move: pixels, channels, convolutions and image features. *chapter checkpoint*

1 Read	2 Write	3 Explain
Open any project from Projects 146-150 and mark setup, rule, and evidence without looking back at the explanation.	Create one sibling program that keeps the same logic but changes the purpose, data, or output wording. *transfer proves learning*	Say what this chapter builds on and where it leads: Next, Chapter 12 (NLP & Text Processing) 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

Recall · Meaning · Structure · Transfer

11.1.Q1 Recall — Bug hunt: "a 7B model fits my 8 GB GPU at fp16" — the load fails. What was left out of the 14 GB?

11.1.Q2 Meaning — Bug hunt: a cost forecast assumed 1 token per word for German legal text and came in 3× under. Why so wrong?

11.1.Q3 Structure — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

11.1.Q4 Transfer — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

11.1.Q5 Recall — What is the most important edge case in Vision: Images as Arrays?

11.1.Q6 Transfer — What proves the skill is transferable?

Answer Key – Chapter 11 – Vision: Images as Arrays

11.1.Q1. Nothing was *left out* — 7B × 2 bytes is already 14 GB of weights alone; 8 GB cannot hold it (int4 quantisation might).

11.1.Q2. Long compound words split into many subtokens — non-English prose often runs 1.5–3 tokens per word; measure with the tokenizer.

11.1.Q3. A — Images are arrays with height, width and channels; vision begins by respecting that shape.

11.1.Q4. A — prediction comes from setup, rule and evidence, not from guessing.

11.1.Q5. Channel order matters: RGB and BGR swaps can silently ruin a model input.

11.1.Q6. Create a grayscale 3x3 image and compute its mean brightness.

Chapter 12 · NLP & Text Processing

Tokenisation, vocabularies, n-grams and text features.

Chapter Skill Map

What this chapter adds: In NLP & Text Processing, the focus is tokenisation, vocabularies, n-grams and text features. You practise it across 10 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 12 assumes the reading habits from Chapter 11: Vision: Images as Arrays. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 13, Embeddings & Similarity Search, reuses this chapter's discipline with a new kind of problem.

1 Projects 151-153
Recognise the pattern and read the code slowly. *name it first*

2 Projects 154-156
Trace similar programs with less prompting and explain what changed.

3 Projects 157-160
Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

NLP & Text Processing: what problem does this solve for a working programmer?

name the move before typing

NLP starts by turning messy human text into tokens, counts and features that code can manipulate.

This chapter's focus is: Tokenisation, vocabularies, n-grams and text features. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
text = "AI helps AI engineers"
tokens = text.lower().split()
vocab = sorted(set(tokens))
print(tokens)
print(vocab)
```

```
output
['ai', 'helps', 'ai', 'engineers']
['ai', 'engineers', 'helps']
```

Lowercasing makes AI and ai the same token before building the vocabulary. *predict first, verify second*

TOKEN

A piece of text treated as one unit, such as a word or subword.

VOCABULARY

The set of known tokens.

N-GRAM

A sequence of n adjacent tokens.

TEXT FEATURE

A numeric representation derived from text.

Beginner model

At beginner level, NLP & Text Processing should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

LLMs use more advanced tokenisers, but the basic contract is still text to token ids.

The edge case to respect

Tokenisation choices change everything: punctuation, casing and subwords affect the model input.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Build bigrams from the token list and count how often each appears.

Example 1

Trace the opening NLP & Text Processing example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read NLP & Text Processing as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Build bigrams from the token list and count how often each appears.

⚠ Common Mistakes

- Copying the surface syntax for NLP & Text Processing without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Tokenisation choices change everything: punctuation, casing and subwords affect the model input.
- Trusting the output before predicting it by hand.

This chapter contains 10 hands-on mini-projects (Projects 151–160). Type each one out, run it, and experiment.

Project 151 Sentiment Analysis with a Pipeline

The Idea Behind It

Transformers power modern NLP and LLMs. Hugging Face makes pretrained models a one-line import. The deeper point is learning how Python turns a small intention like sentiment analysis with a pipeline into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example. *first anchor* Project 152 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

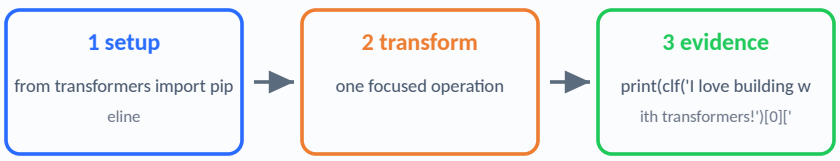
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 152 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sentiment analysis with a pipeline is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from transformers import pipeline` 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: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(clf('I love building with transformers!')[0]['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, 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 transformers import pipeline`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`clf = pipeline('sentiment-analysis')`	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(clf('I love building with transformers!')[0]['label'])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `clf = pipeline('sentiment-analysis')`

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.

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	Sentiment Analysis with a Pipeline: connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: transformers, pipeline, clf, sentiment, analysis. 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: from transformers import pipeline | Core: clf = pipeline('sentiment-analysis') | Evidence: print(clf('I love building with transformers!')[0]['label'])

Reader to Writer Bridge

1. Read: explain sentiment analysis with a pipeline 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(clf('I love building with transformers!')[0]['label'])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from transformers import pipeline` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sentiment analysis with a pipeline and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

pipeline() loads a pretrained model behind a one-line API.

```
from transformers import pipeline

clf = pipeline('sentiment-analysis')
print(clf('I love building with transformers!')[0]['label'])
```

```
output
POSITIVE
```

How it works:

1. Start at `from transformers import pipeline`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(clf('I love building with transformers!')[0]['label'])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a different pretrained model.

⚠ Common Mistakes

- Forgetting models download on first use.
- Mismatched tokenizer and model checkpoints.

Skills gained `predicting exact output`

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: `locate the data`, `name the transformation`, and `verify the output`. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: `set up the facts` -> `apply one focused rule` -> `make the result observable`. *tiny program, full habit*

Project 152 Sub-Word Tokenization

💡 The Idea Behind It

`Transformers power modern NLP and LLMs. Hugging Face makes pretrained models a one-line import.` The deeper point is learning how Python turns a small intention like sub-word tokenization into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen `code reading`: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, `NLP & Text Processing`, to a concrete mini-program.
- Read the program as `input or setup`, `transformation`, then `output`. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 151, where you practised sentiment analysis with a pipeline, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 153 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

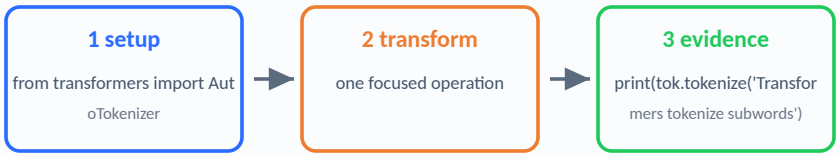
`Skill step:` adds one small variation while the chapter pattern is still new. It builds on Project 151: sentiment analysis with a pipeline. Project 153 changes the surface problem while preserving the underlying method. *progress check*

`Reading target:` sub-word tokenization is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `from transformers import AutoTokenizer` 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: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(tok.tokenize('Transformers tokenize subwords'))` exposes the result; match it 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 transformers import AutoTokenizer`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`tok = AutoTokenizer.from_pretrained('bert-base-uncased')`	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(tok.tokenize('Transformers tokenize subwords'))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `tok = AutoTokenizer.from_pretrained('bert-base-uncased')`

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. 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	Sub-Word Tokenization: connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: transformers, AutoTokenizer, tok, from_pretrained, bert. 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: from transformers import AutoTokenizer | Core: tok = AutoTokenizer.from_pretrained('bert-base-uncased') | Evidence: print(tok.tokenize('Transformers tokenize subwords'))

🛠️ Reader to Writer Bridge

1. Read: explain sub-word tokenization in one sentence before touching the 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(tok.tokenize('Transformers tokenize subwords'))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from transformers import AutoTokenizer` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sub-word tokenization and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

WordPiece splits rare words into known sub-word units (## prefix).

```python
from transformers import AutoTokenizer

tok = AutoTokenizer.from_pretrained('bert-base-uncased')
print(tok.tokenize('Transformers tokenize subwords'))
```

```
output
['transformers', 'token', '##ize', 'sub', '##words']
```

How it works:

1. Start at `from transformers import AutoTokenizer`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(tok.tokenize('Transformers tokenize subwords'))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a different pretrained model.

⚠ Common Mistakes

- Forgetting models download on first use.
- Mismatched tokenizer and model checkpoints.

Skills gained predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 153 Scaled Dot-Product Attention from Scratch

The Idea Behind It

Transformers power modern NLP and LLMs. Hugging Face makes pretrained models a one-line import. The deeper point is learning how Python turns a small intention like scaled dot-product attention from scratch into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 152, where you practised sub-word tokenization, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 154 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 152: sub-word tokenization. Project 154 changes the surface problem while preserving the underlying method.

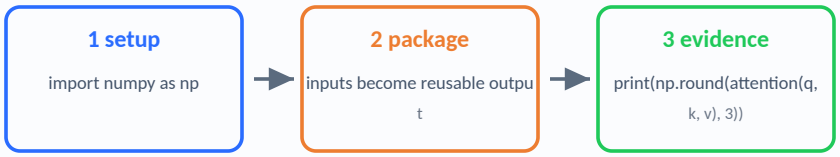
progress check

Reading target: scaled dot-product attention from scratch is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` 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: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(np.round(attention(q, k, v), 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, package applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`return w @ 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(np.round(attention(q, k, v), 3))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: NLP & Text Processing becomes practical when you can build the syntax 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 @ 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. Brackets: Signal indexing, lists/arrays, table-like access, or type shapes. Colon: Opens Python blocks, separates keys from values, or 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	Scaled Dot-Product Attention from Scratch: connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.

Layer	What to read
Names	Watch these names first: numpy, np, attention, scores, sqrt. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: return w @ V | Evidence: print(np.round(attention(q, k, v), 3))

🛠️ Reader to Writer Bridge

1. Read: explain scaled dot-product attention from scratch in one sentence before touching the 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(np.round(attention(q, k, v), 3))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of scaled dot-product attention from scratch and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format.

same idea, new surface

🤖 From Concept to Code (Python)

Attention weights values by the softmax of query·key similarity.

```python
import numpy as np

def attention(Q, K, V):
    scores = Q @ K.T / np.sqrt(K.shape[-1])
    w = np.exp(scores) / np.exp(scores).sum(-1, keepdims=True)
    return w @ V

q = k = v = np.eye(2)
print(np.round(attention(q, k, v), 3))
```

```
output
[[0.67 0.33]
 [0.33 0.67]]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(attention(q, k, v), 3))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a different pretrained model.

⚠ Common Mistakes

- Forgetting models download on first use.
- Mismatched tokenizer and model checkpoints.

Skills gained packaging logic into reusable steps | choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 154 Text Generation with GPT-2

The Idea Behind It

Transformers power modern NLP and LLMs. Hugging Face makes pretrained models a one-line import. The deeper point is learning how Python turns a small intention like text generation with gpt-2 into a readable

sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 153, where you practised scaled dot-product attention from scratch, and asks you to reuse the same reasoning pattern. reuse the pattern Project 155 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

🧭 Skill Roadmap

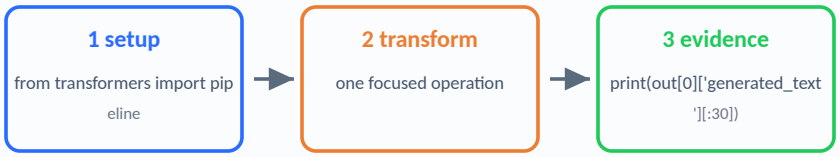
Skill step: moves you from recognition toward independent tracing. It builds on Project 153: scaled dot-product attention from scratch. Project 155 changes the surface problem while preserving the underlying method. progress check

Reading target: text generation with gpt-2 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `from transformers import pipeline` 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: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(out[0]['generated_text'][:30])` exposes the result; match it 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 transformers import pipeline`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`gen = pipeline('text-generation', model='gpt2')`	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[0]['generated_text'][:30])`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `gen = pipeline('text-generation', model='gpt2')`

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.

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	Text Generation with GPT-2: connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: transformers, pipeline, gen, text, generation. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.

Layer	What to read
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: from transformers import pipeline | Core: gen = pipeline('text-generation', model='gpt2') | Evidence: print(out[0]['generated_text'][:30])

🛠️ Reader to Writer Bridge

1. Read: explain text generation with gpt-2 in one sentence before touching the 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(out[0]['generated_text'][:30])`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from transformers import pipeline` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of text generation with gpt-2 and name what must stay the 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 autoregressive model continues the prompt token by token.

```python
from transformers import pipeline

gen = pipeline('text-generation', model='gpt2')
out = gen('AI engineering is', max_new_tokens=5)
print(out[0]['generated_text'][:30])
```

```
output
AI engineering is the future of
```

How it works:

1. Start at `from transformers import pipeline`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*

2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(out[0]['generated_text'][:30])`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a different pretrained model.

⚠ Common Mistakes

- Forgetting models download on first use.
- Mismatched tokenizer and model checkpoints.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 155 Sentence Embeddings

💡 The Idea Behind It

Transformers power modern NLP and LLMs. Hugging Face makes pretrained models a one-line import. The deeper point is learning how Python turns a small intention like sentence embeddings into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*

- This project moves you from recognition toward independent tracing.
- It builds on Project 154, where you practised text generation with gpt-2, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 156 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 154: text generation with gpt-2. Project 156 changes the surface problem while preserving the underlying method.

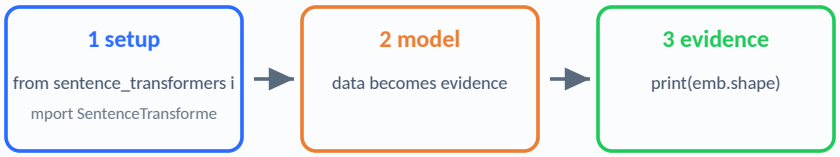
progress check

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

Logic Lens

- Setup: `from sentence_transformers import SentenceTransformer` 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: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(emb.shape)` exposes the result; match it 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	`from sentence_transformers import SentenceTransformer`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.

Move	Code to inspect	Why it matters
2. Apply the rule	`model = SentenceTransformer('all-MiniLM-L6-v2')`	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(emb.shape)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `model = SentenceTransformer('all-MiniLM-L6-v2')`

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. 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	Sentence Embeddings: connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: sentence_transformers, SentenceTransformer, model, all, MiniLM. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: from sentence_transformers import SentenceTransformer | Core: model = SentenceTransformer('all-MiniLM-L6-v2') | Evidence: print(emb.shape)

Reader to Writer Bridge

1. Read: explain sentence embeddings in one sentence before touching the 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(emb.shape)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `from sentence_transformers import SentenceTransformer` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sentence embeddings and name what must stay the 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 sentence becomes a fixed-length vector for search and clustering.

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
emb = model.encode(['hello world'])
print(emb.shape)
```

```
output
(1, 384)
```

How it works:

1. Start at `from sentence_transformers import SentenceTransformer`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(emb.shape)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Try a different pretrained model.

⚠ Common Mistakes

- Forgetting models download on first use.
- Mismatched tokenizer and model checkpoints.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 156 Binary Cross-Entropy (set 24)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 155, where you practised sentence embeddings, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 157 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 155: sentence embeddings. Project 157 changes the surface problem while preserving the underlying method.

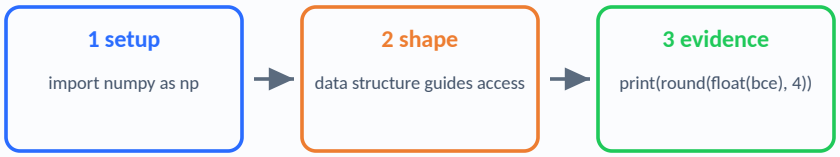
progress check

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`y = np.array([1, 0, 1])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `y = np.array([1, 0, 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. 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	Binary Cross-Entropy (set 24): connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([1, 0, 1]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([1, 0, 1])
```

```
p = np.array([0.7, 0.2, 0.5])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 157 Sigmoid of 4.5

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of 4.5 into a

readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project pushes the same idea into a less automatic situation.
- It builds on Project 156, where you practised binary cross-entropy, and asks you to reuse the same reasoning pattern. reuse the pattern Project 158 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

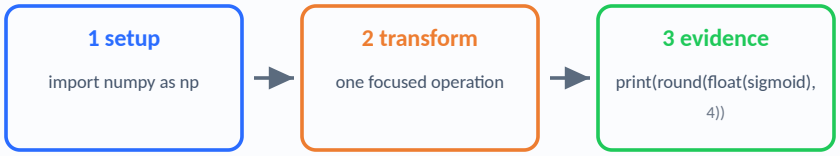
Skill step: pushes the same idea into a less automatic situation. It builds on Project 156: binary cross-entropy. Project 158 changes the surface problem while preserving the underlying method. progress check

Reading target: sigmoid of 4.5 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to make one intention visible in code. It adds that move to the chapter skill: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(round(float(sigmoid), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of 4.5: connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

Reader to Writer Bridge

1. Read: explain sigmoid of 4.5 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 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of 4.5 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)

The sigmoid squashes any real number into (0, 1).

```
import numpy as np

z = 4.5
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.989
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 158 Softmax of a 3-Vector (set 32)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 157, where you practised sigmoid of 4.5, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 159 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

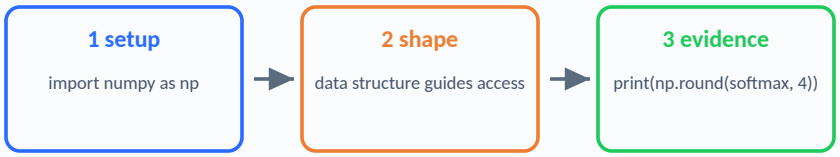
Skill step: pushes the same idea into a less automatic situation. It builds on Project 157: sigmoid of 4.5. Project 159 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(np.round(softmax, 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`z = np.array([-2.5, 0.0, 2.5])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `z = np.array([-2.5, 0.0, 2.5])`

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. 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	Softmax of a 3-Vector (set 32): connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([-2.5, 0.0, 2.5]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```
import numpy as np

z = np.array([-2.5, 0.0, 2.5])
```

```
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0062 0.0754 0.9184]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 159 Dot Product (set 33)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a

readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project pushes the same idea into a less automatic situation.
- It builds on Project 158, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. reuse the pattern Project 160 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

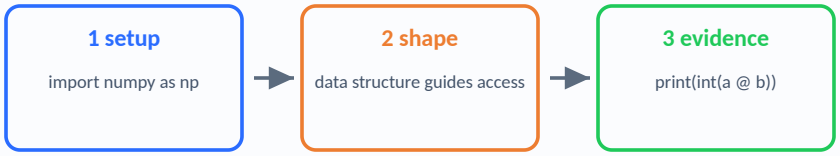
Skill step: pushes the same idea into a less automatic situation. It builds on Project 158: softmax of a 3-vector. Project 160 changes the surface problem while preserving the underlying method. progress check

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

Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(int(a @ 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`a = np.array([6, 9, 3])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: NLP & Text Processing becomes practical when you can build the syntax 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 = np.array([6, 9, 3])`

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. 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	Dot Product (set 33): connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([6, 9, 3]) | Evidence: print(int(a @ b))

Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([6, 9, 3])
b = np.array([2, 7, 3])
print(int(a @ b))
```

```
output
84
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 160 Binary Cross-Entropy (set 34)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, NLP & Text Processing, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 159, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

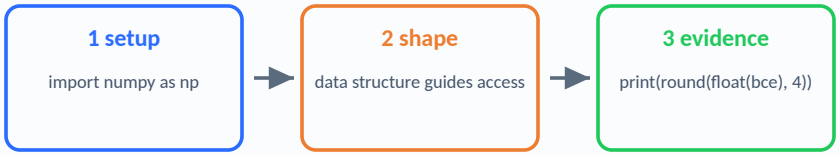
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 159: dot product. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` gives the program its first fact, tool, or entry point. Use it as the anchor for the later logic.
- Core logic: The important move is whole-array thinking: one expression changes many values at once. This builds the habit of reading vectorised code as a mathematical operation, not as a hidden loop.
- Writing move: this project practises how to choose a data shape before writing the operation. It adds that move to the chapter skill: tokenisation, vocabularies, n-grams and text features.
- Evidence: `print(round(float(bce), 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, 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: tokenisation, vocabularies, n-grams and text features.
2. Apply the rule	`y = np.array([1, 0, 1])`	This line is the vectorised move. Read it as one mathematical operation applied across many values. *slow line*
3. Check the evidence	`print(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `y = np.array([1, 0, 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. 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	Binary Cross-Entropy (set 34): connect the surface task to the chapter skill, tokenisation, vocabularies, n-grams and text features.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([1, 0, 1]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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, 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([1, 0, 1])
p = np.array([0.3, 0.6, 0.1])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
1.4743
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Chapter Progress Checkpoint

What this chapter added: after 10 mini-projects, NLP & Text Processing should feel like a reusable Python move: tokenisation, vocabularies, n-grams and text features. *chapter checkpoint*

1 Read	2 Write	3 Explain
Open any project from Projects 151-160 and mark setup, rule, and evidence without looking back at the explanation.	Create one sibling program that keeps the same logic but changes the purpose, data, or output wording. *transfer proves learning*	Say what this chapter builds on and where it leads: Next, Chapter 13 (Embeddings & Similarity Search) 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

Recall · Meaning · Structure · Transfer

12.1.Q1 Recall — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

12.1.Q2 Meaning — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

12.1.Q3 Structure — What is the most important edge case in NLP & Text Processing?

12.1.Q4 Transfer What proves the skill is transferable?

Answer Key – Chapter 12 – NLP & Text Processing

12.1.Q1. A — NLP starts by turning messy human text into tokens, counts and features that code can manipulate.

12.1.Q2. A — prediction comes from setup, rule and evidence, not from guessing.

12.1.Q3. Tokenisation choices change everything: punctuation, casing and subwords affect the model input.

12.1.Q4. Build bigrams from the token list and count how often each appears.

Chapter 13 · Embeddings & Similarity Search

Vector representations and nearest-neighbour retrieval.

Chapter Skill Map

What this chapter adds: In Embeddings & Similarity Search, the focus is vector representations and nearest-neighbour retrieval. You practise it across 5 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 13 assumes the reading habits from Chapter 12: NLP & Text Processing. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 14, Transformers & Attention from Scratch, reuses this chapter's discipline with a new kind of problem.

1 **Project 161** Recognise the pattern and read the code slowly. *name it first*	2 **Project 162** Trace similar programs with less prompting and explain what changed.	3 **Projects 163-165** Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Embeddings & Similarity Search: what problem does this solve for a working programmer?

name the move before typing

Embeddings turn items into vectors so similarity search can retrieve by meaning-like distance.

This chapter's focus is: Vector representations and nearest-neighbour retrieval. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```python
import numpy as np

a = np.array([1.0, 0.0])
b = np.array([0.8, 0.6])
cosine = (a @ b) / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cosine), 2))
```

```
output
0.8
```

The vectors point in similar but not identical directions, so cosine is 0.8. *predict first, verify second*

EMBEDDING

A vector representation of text, image or other data.

COSINE SIMILARITY

A direction-based similarity score between vectors.

NEAREST NEIGHBOUR

The stored vector closest to a query vector.

VECTOR INDEX

A data structure built for fast similarity search.

Beginner model

At beginner level, Embeddings & Similarity Search should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Search, recommendations and RAG all rely on the same vector-neighbour idea.

The edge case to respect

Embedding scale, normalisation and stale indexes can all change retrieval quality.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Add a third vector and rank all candidates by cosine similarity.

Example 1

Trace the opening Embeddings & Similarity Search example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read Embeddings & Similarity Search as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Add a third vector and rank all candidates by cosine similarity.

⚠ Common Mistakes

- Copying the surface syntax for Embeddings & Similarity Search without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Embedding scale, normalisation and stale indexes can all change retrieval quality.
- Trusting the output before predicting it by hand.

This chapter contains 5 hands-on mini-projects (Projects 161–165). Type each one out, run it, and experiment.

Project 161 Cosine Similarity (set 35)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Embeddings & Similarity Search, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 162 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

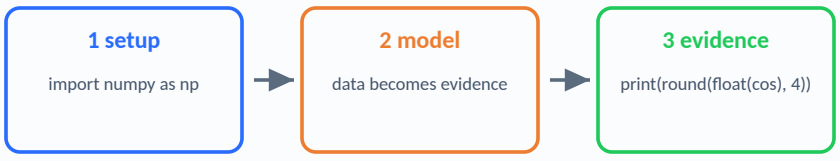
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 162 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` 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: vector representations and nearest-neighbour retrieval.
- Evidence: `print(round(float(cos), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vector representations and nearest-neighbour retrieval.
2. Apply the rule	`a = np.array([7, 2, 4], dtype=float)`	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(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Embeddings & Similarity Search becomes practical when you can build the syntax 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 = np.array([7, 2, 4], dtype=float)`

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. 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	Cosine Similarity (set 35): connect the surface task to the chapter skill, vector representations and nearest-neighbour retrieval.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([7, 2, 4], dtype=float) | Evidence: print(round(float(cos), 4))

Reader to Writer Bridge

1. Read: explain cosine 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 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([7, 2, 4], dtype=float)
b = np.array([5, 6, 7], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.8609
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 162 Sigmoid of -5.75

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -5.75 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Embeddings & Similarity Search, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 161, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 163 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

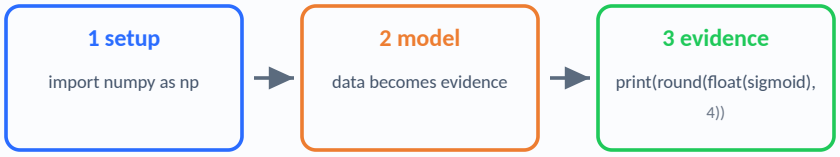
Skill step: moves you from recognition toward independent tracing. It builds on Project 161: cosine similarity. Project 163 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of -5.75 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` 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: vector representations and nearest-neighbour retrieval.
- Evidence: `print(round(float(sigmoid), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vector representations and nearest-neighbour retrieval.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	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(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -5.75: connect the surface task to the chapter skill, vector representations and nearest-neighbour retrieval.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of -5.75 in one sentence before touching the 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -5.75 and name what must stay the 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 sigmoid squashes any real number into (0, 1).

```python
import numpy as np

z = -5.75
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.0032
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 163 Softmax of a 3-Vector (set 37)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Embeddings & Similarity Search, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 162, where you practised sigmoid of -5.75, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 164 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

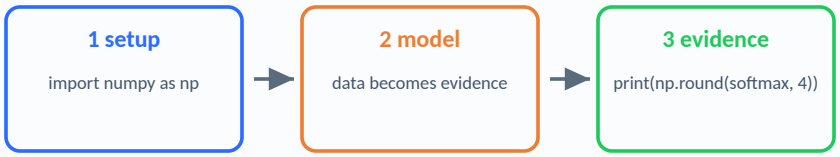
Skill step: moves you from recognition toward independent tracing. It builds on Project 162: sigmoid of -5.75. Project 164 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` 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: vector representations and nearest-neighbour retrieval.
- Evidence: `print(np.round(softmax, 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vector representations and nearest-neighbour retrieval.
2. Apply the rule	`z = np.array([-4.0, -1.5, 1.0])`	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(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

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

CORE LINE `z = np.array([-4.0, -1.5, 1.0])`

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. 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

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	Softmax of a 3-Vector (set 37): connect the surface task to the chapter skill, vector representations and nearest-neighbour retrieval.

Layer	What to read
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([-4.0, -1.5, 1.0]) | Evidence: print(np.round(softmax, 4))

🛠️ Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```python
import numpy as np

z = np.array([-4.0, -1.5, 1.0])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.0062 0.0754 0.9184]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 164 Dot Product (set 38)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Embeddings & Similarity Search, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 163, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 165 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

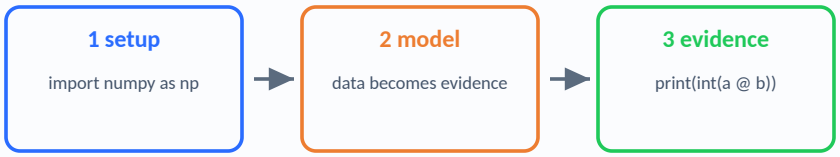
Skill step: pushes the same idea into a less automatic situation. It builds on Project 163: softmax of a 3-vector. Project 165 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` 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: vector representations and nearest-neighbour retrieval.
- Evidence: `print(int(a @ 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vector representations and nearest-neighbour retrieval.
2. Apply the rule	`a = np.array([2, 5, 8])`	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(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Embeddings & Similarity Search becomes practical when you can build the syntax 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 = np.array([2, 5, 8])`

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. 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	Dot Product (set 38): connect the surface task to the chapter skill, vector representations and nearest-neighbour retrieval.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([2, 5, 8]) | Evidence: print(int(a @ b))

Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([2, 5, 8])
b = np.array([3, 8, 4])
print(int(a @ b))
```

```
output
78
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 165 Binary Cross-Entropy (set 39)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Embeddings & Similarity Search, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 164, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

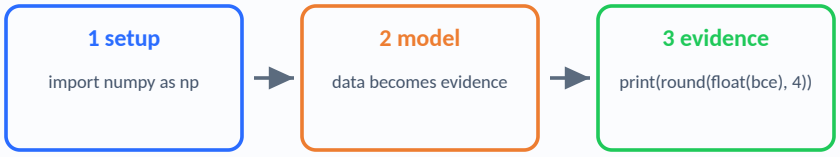
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 164: dot product. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` 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: vector representations and nearest-neighbour retrieval.
- Evidence: `print(round(float(bce), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: vector representations and nearest-neighbour retrieval.
2. Apply the rule	`y = np.array([0, 1, 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(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

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

CORE LINE `y = np.array([0, 1, 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. 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

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	Binary Cross-Entropy (set 39): connect the surface task to the chapter skill, vector representations and nearest-neighbour retrieval.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([0, 1, 0]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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, 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([0, 1, 0])
p = np.array([0.5, 0.8, 0.3])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Chapter Progress Checkpoint

What this chapter added: after 5 mini-projects, Embeddings & Similarity Search should feel like a reusable Python move: vector representations and nearest-neighbour retrieval. *chapter checkpoint*

1 Read	2 Write	3 Explain
Open any project from Projects 161-165 and mark setup, rule, and evidence without looking back at the explanation.	Create one sibling program that keeps the same logic but changes the purpose, data, or output wording. *transfer proves learning*	Say what this chapter builds on and where it leads: Next, Chapter 14 (Transformers & Attention from Scratch) 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

Recall · Meaning · Structure · Transfer

13.1.Q1 Meaning — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

13.1.Q2 Meaning — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

13.1.Q3 Structure — What is the most important edge case in Embeddings & Similarity Search?

13.1.Q4 Transfer What proves the skill is transferable?

Answer Key – Chapter 13 – Embeddings & Similarity Search

13.1.Q1. A — Embeddings turn items into vectors so similarity search can retrieve by meaning-like distance.

13.1.Q2. A — prediction comes from setup, rule and evidence, not from guessing.

13.1.Q3. Embedding scale, normalisation and stale indexes can all change retrieval quality.

13.1.Q4. Add a third vector and rank all candidates by cosine similarity.

Chapter 14 · Transformers & Attention from Scratch

Queries, keys, values — attention computed by hand.

Chapter Skill Map

What this chapter adds: In Transformers & Attention from Scratch, the focus is queries, keys, values — attention computed by hand. You practise it across 23 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 14 assumes the reading habits from Chapter 13: Embeddings & Similarity Search. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 15, LLM Engineering, Prompts & RAG, reuses this chapter's discipline with a new kind of problem.

1 **Projects 166-172**	2 **Projects 173-179**	3 **Projects 180-188**
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

Transformers & Attention from Scratch: what problem does this solve for a working programmer?

name the move before typing

Attention scores how strongly one token should read information from other tokens.

This chapter's focus is: Queries, keys, values — attention computed by hand. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
import numpy as np

q = np.array([1.0, 0.0])
keys = np.array([[1.0, 0.0], [0.0, 1.0]])
scores = keys @ q
weights = np.exp(scores) / np.exp(scores).sum()
print(np.round(weights, 2))
```

```
output
[0.73 0.27]
```

The first key matches the query better, so it receives the larger weight. *predict first, verify second*

QUERY

The vector asking what information is needed.

KEY

The vector used to score whether a token is relevant.

VALUE

The information vector mixed according to attention weights.

SOFTMAX

A function turning scores into a probability-like distribution.

Beginner model

At beginner level, Transformers & Attention from Scratch should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Transformers scale this small query-key-value move across heads, layers and tokens.

The edge case to respect

Self-attention cost grows with sequence length squared, so long context is expensive.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Add a value matrix and compute the weighted sum after softmax.

Example 1

Trace the opening Transformers & Attention from Scratch example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

⚡ Mental-Math Shortcut

Read Transformers & Attention from Scratch as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Add a value matrix and compute the weighted sum after softmax.

⚠ Common Mistakes

- Copying the surface syntax for Transformers & Attention from Scratch without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Self-attention cost grows with sequence length squared, so long context is expensive.
- Trusting the output before predicting it by hand.

This chapter contains 23 hands-on mini-projects (Projects 166–188). Type each one out, run it, and experiment.

Project 166 Multi-Head Split (d_model=128)

💡 The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 167 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

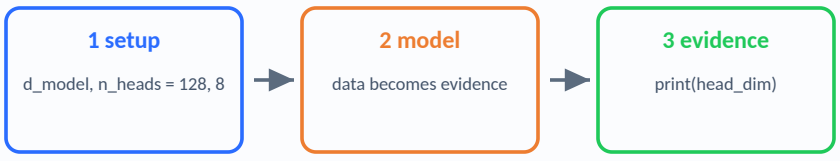
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 167 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `d_model, n_heads = 128, 8` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 128, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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

Code Reading Stack

Layer	What to read
Intent	Multi-Head Split (d_model=128): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. 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: d_model, n_heads = 128, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

Reader to Writer Bridge

1. Read: explain multi-head split 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 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 128, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```
d_model, n_heads = 128, 8
assert d_model % n_heads == 0, 'must divide evenly'
head_dim = d_model // n_heads
print(head_dim)
```

```
output
16
```

How it works:

1. Start at `d_model, n_heads = 128, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 167 Count Parameters (19->14->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 166, where you practised multi-head split, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 168 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 166: multi-head split. Project 168 changes the surface problem while preserving the underlying method. *progress check*

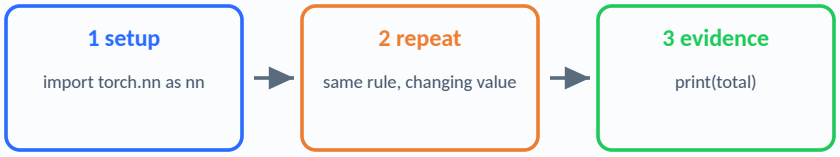
Reading target: count parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import torch.nn as nn` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	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(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (19->14->3 MLP): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: torch, nn, net, Sequential, Linear. 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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

🛠️ Reader to Writer Bridge

1. Read: explain count 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 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```
import torch.nn as nn

net = nn.Sequential(nn.Linear(19, 14), nn.ReLU(), nn.Linear(14, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
325
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 168 Attention Cost for seq_len=2048

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=2048 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 167, where you practised count parameters, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 169 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 167: count parameters. Project 169 changes the surface problem while preserving the underlying method.

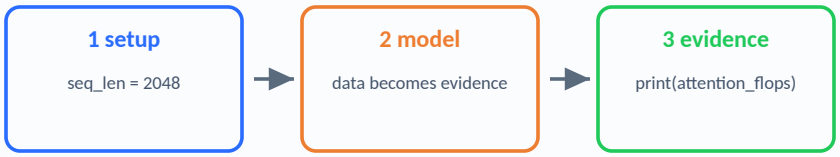
progress check

Reading target: attention cost for seq_len=2048 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `seq_len = 2048` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 2048`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=2048: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.

Layer	What to read
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: seq_len = 2048 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

🛠️ Reader to Writer Bridge

1. Read: explain attention cost for seq_len=2048 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 2048` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=2048 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 2048
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
4194304
```

How it works:

1. Start at `seq_len = 2048`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 169 LLM Cost for 16000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 16000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.

- It builds on Project 168, where you practised attention cost for seq_len=2048, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 170 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 168: attention cost for seq_len=2048. Project 170 changes the surface problem while preserving the underlying method.

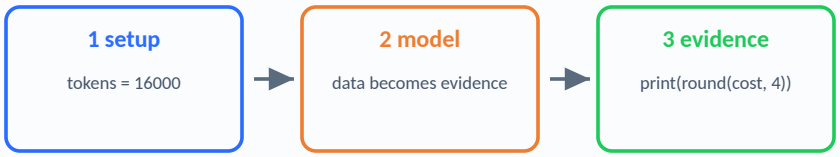
progress check

Reading target: llm cost for 16000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `tokens = 16000` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(cost, 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, 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	`tokens = 16000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something

Move	Code to inspect	Why it matters
		measurable. slow line
3. Check the evidence	`print(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 16000 Tokens: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 16000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

Reader to Writer Bridge

1. Read: explain llm cost for 16000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 16000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 16000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 16000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.008
```

How it works:

1. Start at `tokens = 16000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 170 Multi-Head Split (d_model=136)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 169, where you practised llm cost for 16000 tokens, and asks you to reuse the same reasoning pattern. reuse the pattern Project 171 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 169: llm cost for 16000 tokens. Project 171 changes the surface problem while preserving the underlying method.

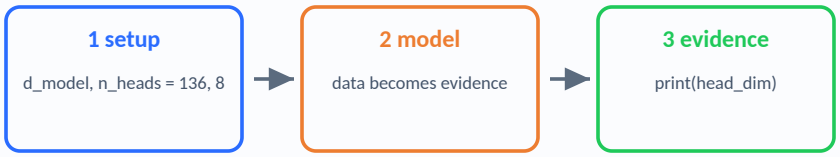
progress check

Reading target: multi-head split is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `d_model, n_heads = 136, 8` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 136, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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

Code Reading Stack

Layer	What to read
Intent	Multi-Head Split (d_model=136): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. 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: d_model, n_heads = 136, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 136, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```
d_model, n_heads = 136, 8
assert d_model % n_heads == 0, 'must divide evenly'
```

```
head_dim = d_model // n_heads
print(head_dim)
```

```
output
17
```

How it works:

1. Start at `d_model, n_heads = 136, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 171 Count Parameters (20->12->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of

setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 170, where you practised multi-head split, and asks you to reuse the same reasoning pattern. reuse the pattern Project 172 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

🧭 Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 170: multi-head split. Project 172 changes the surface problem while preserving the underlying method. progress check

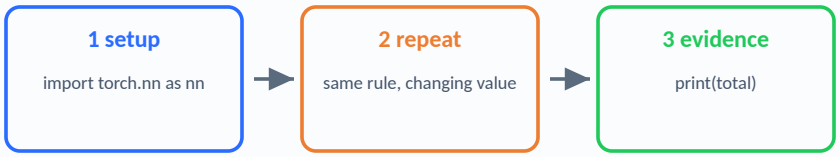
Reading target: count parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import torch.nn as nn` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	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(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (20->12->3 MLP): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: torch, nn, net, Sequential, Linear. Names are the reader's map of roles and responsibilities.

Layer	What to read
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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

🛠️ Reader to Writer Bridge

1. Read: explain count 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 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```python
import torch.nn as nn

net = nn.Sequential(nn.Linear(20, 12), nn.ReLU(), nn.Linear(12, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
291
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*

2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 172 Attention Cost for seq_len=5760

💡 The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=5760 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*

- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 171, where you practised count parameters, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 173 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

`Skill step:` adds one small variation while the chapter pattern is still new. It builds on Project 171: count parameters. Project 173 changes the surface problem while preserving the underlying method.

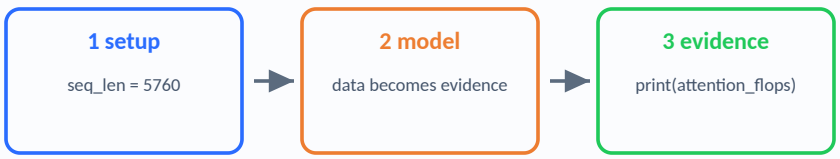
progress check

`Reading target:` attention cost for seq_len=5760 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- `Setup:` `seq_len = 5760` 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: queries, keys, values — attention computed by hand.
- `Evidence:` `print(attention_flops)` exposes the result; match it 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	`seq_len = 5760`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: `queries, keys, values — attention computed by hand`.

Move	Code to inspect	Why it matters
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=5760: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. 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: seq_len = 5760 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

Reader to Writer Bridge

1. Read: explain attention cost for seq_len=5760 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 5760` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=5760 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 5760
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
33177600
```

How it works:

1. Start at `seq_len = 5760`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.

- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 173 LLM Cost for 45000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 45000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 172, where you practised attention cost for seq_len=5760, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 174 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 172: attention cost for seq_len=5760. Project 174 changes the surface problem while preserving the underlying method.

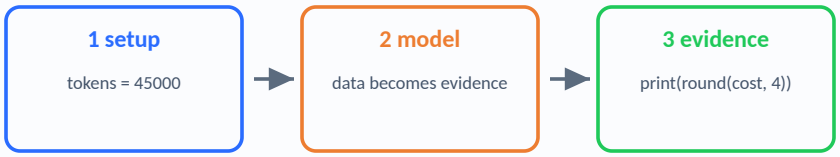
progress check

Reading target: llm cost for 45000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `tokens = 45000` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(cost, 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, 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	`tokens = 45000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	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(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 45000 Tokens: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 45000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

🛠️ Reader to Writer Bridge

1. Read: explain llm cost for 45000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 45000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 45000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 45000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.0225
```

How it works:

1. Start at `tokens = 45000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 174 Multi-Head Split (d_model=368)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 173, where you practised llm cost for 45000 tokens, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 175 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

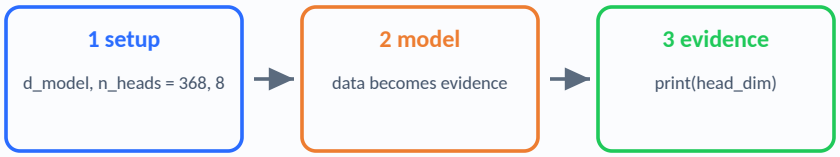
Skill step: moves you from recognition toward independent tracing. It builds on Project 173: llm cost for 45000 tokens. Project 175 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `d_model, n_heads = 368, 8` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 368, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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

Code Reading Stack

Layer	What to read
Intent	Multi-Head Split (d_model=368): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. Names are the reader's map of roles and responsibilities.

Layer	What to read
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: d_model, n_heads = 368, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 368, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```
d_model, n_heads = 368, 8
assert d_model % n_heads == 0, 'must divide evenly'
head_dim = d_model // n_heads
print(head_dim)
```

```
output
46
```

How it works:

1. Start at `d_model, n_heads = 368, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 175 Count Parameters (49->14->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 174, where you practised multi-head split, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 176 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

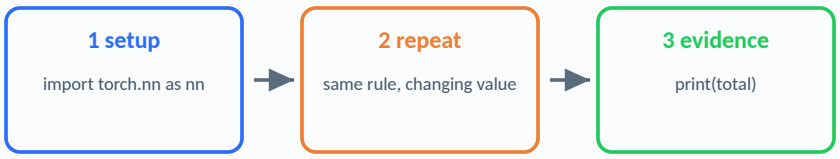
Skill step: moves you from recognition toward independent tracing. It builds on Project 174: multi-head split. Project 176 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count 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 torch.nn as nn` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (49->14->3 MLP): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: torch, nn, net, Sequential, Linear. 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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

Reader to Writer Bridge

1. Read: explain count 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 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```python
import torch.nn as nn

net = nn.Sequential(nn.Linear(49, 14), nn.ReLU(), nn.Linear(14, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
745
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 176 Attention Cost for seq_len=5888

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=5888 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 175, where you practised count parameters, and asks you to reuse the same reasoning pattern. reuse the pattern Project 177 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 175: count parameters. Project 177 changes the surface problem while preserving the underlying method.

progress check

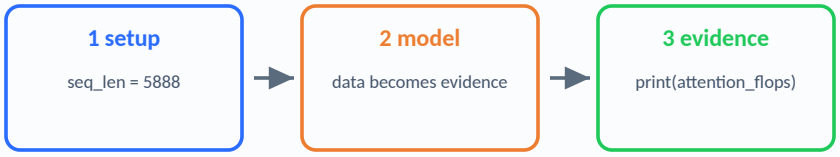
Reading target: attention cost for seq_len=5888 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `seq_len = 5888` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 5888`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=5888: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. 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: seq_len = 5888 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

🛠️ Reader to Writer Bridge

1. Read: explain attention cost for seq_len=5888 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 5888` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=5888 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 5888
attention_flops = seq_len ** 2  # pairwise scores
```

```
print(attention_flops)
```

```
output
34668544
```

How it works:

1. Start at `seq_len = 5888`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 177 LLM Cost for 46000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 46000 tokens into a readable

sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 176, where you practised attention cost for seq_len=5888, and asks you to reuse the same reasoning pattern. reuse the pattern Project 178 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

🧭 Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 176: attention cost for seq_len=5888. Project 178 changes the surface problem while preserving the underlying method.

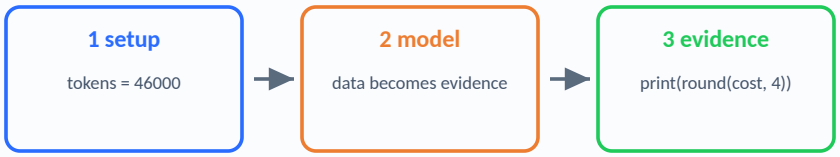
progress check

Reading target: llm cost for 46000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `tokens = 46000` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(cost, 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, 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	`tokens = 46000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	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(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 46000 Tokens: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: tokens, price_per_1k, cost, round. Names are the reader's map of roles and responsibilities.

Layer	What to read
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: tokens = 46000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

🛠️ Reader to Writer Bridge

1. Read: explain llm cost for 46000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 46000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 46000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 46000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.023
```

How it works:

1. Start at `tokens = 46000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 178 Count Parameters (4->8->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 177, where you practised llm cost for 46000 tokens, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 179 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

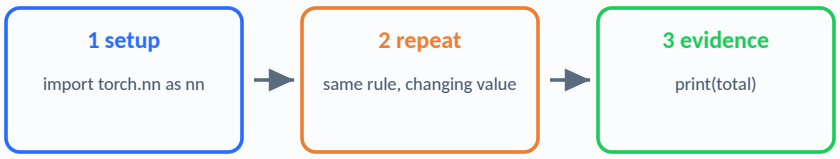
Skill step: moves you from recognition toward independent tracing. It builds on Project 177: llm cost for 46000 tokens. Project 179 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count 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 torch.nn as nn` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (4->8->3 MLP): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: torch, nn, net, Sequential, Linear. 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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

Reader to Writer Bridge

1. Read: explain count 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 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```python
import torch.nn as nn

net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
67
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 179 3 SGD Steps (lr=0.02)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like 3 sgd steps into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 178, where you practised count parameters, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 180 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 178: count parameters. Project 180 changes the surface problem while preserving the underlying method.

progress check

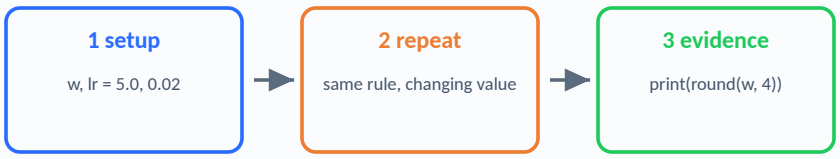
Reading target: 3 sgd steps is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `w, lr = 5.0, 0.02` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(w, 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	`w, lr = 5.0, 0.02`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`for _ in range(3):`	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(round(w, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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 _ in range(3):`

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. Colon: Opens Python blocks, separates keys from values, or 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	3 SGD Steps (lr=0.02): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: lr, range, grad, round. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: w, lr = 5.0, 0.02 | Core: for _ in range(3): | Evidence: print(round(w, 4))

🛠️ Reader to Writer Bridge

1. Read: explain 3 sgd 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 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(round(w, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `w, lr = 5.0, 0.02` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of 3 sgd 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)

Smaller learning rates converge more slowly but more stably.

```
w, lr = 5.0, 0.02
for _ in range(3):
    grad = 2 * (w - 1)
```

```
    w -= lr * grad
print(round(w, 4))
```

```
output
4.5389
```

How it works:

1. Start at `w, lr = 5.0, 0.02`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(w, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 180 Attention Cost for seq_len=128

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=128 into a readable

sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project moves you from recognition toward independent tracing.
- It builds on Project 179, where you practised 3 sgd steps, and asks you to reuse the same reasoning pattern. reuse the pattern Project 181 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

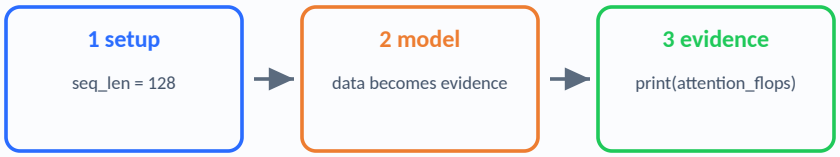
Skill step: moves you from recognition toward independent tracing. It builds on Project 179: 3 sgd steps. Project 181 changes the surface problem while preserving the underlying method. progress check

Reading target: attention cost for seq_len=128 is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `seq_len = 128` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 128`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=128: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.

Layer	What to read
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: seq_len = 128 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

🛠️ Reader to Writer Bridge

1. Read: explain attention cost for seq_len=128 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 128` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=128 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 128
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
16384
```

How it works:

1. Start at `seq_len = 128`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 181 LLM Cost for 1000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 1000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.

- It builds on Project 180, where you practised attention cost for seq_len=128, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 182 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 180: attention cost for seq_len=128. Project 182 changes the surface problem while preserving the underlying method.

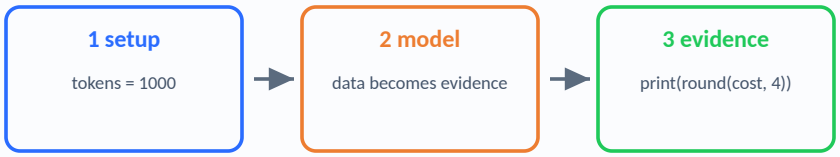
progress check

Reading target: llm cost for 1000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `tokens = 1000` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(cost, 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, 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	`tokens = 1000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something

Move	Code to inspect	Why it matters
		measurable. *slow line*
3. Check the evidence	`print(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 1000 Tokens: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 1000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

Reader to Writer Bridge

1. Read: explain llm cost for 1000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 1000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 1000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 1000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.0005
```

How it works:

1. Start at `tokens = 1000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 182 Multi-Head Split (d_model=16)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project pushes the same idea into a less automatic situation.
- It builds on Project 181, where you practised llm cost for 1000 tokens, and asks you to reuse the same reasoning pattern. reuse the pattern Project 183 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

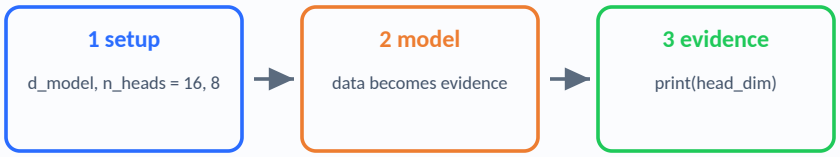
Skill step: pushes the same idea into a less automatic situation. It builds on Project 181: llm cost for 1000 tokens. Project 183 changes the surface problem while preserving the underlying method. progress check

Reading target: multi-head split is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `d_model, n_heads = 16, 8` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 16, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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

Code Reading Stack

Layer	What to read
Intent	Multi-Head Split (d_model=16): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. 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: d_model, n_heads = 16, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 16, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```
d_model, n_heads = 16, 8
assert d_model % n_heads == 0, 'must divide evenly'
```

```
head_dim = d_model // n_heads
print(head_dim)
```

```
output
2
```

How it works:

1. Start at `d_model, n_heads = 16, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 183 Count Parameters (5->18->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of

setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project pushes the same idea into a less automatic situation.
- It builds on Project 182, where you practised multi-head split, and asks you to reuse the same reasoning pattern. reuse the pattern Project 184 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

🧭 Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 182: multi-head split. Project 184 changes the surface problem while preserving the underlying method. progress check

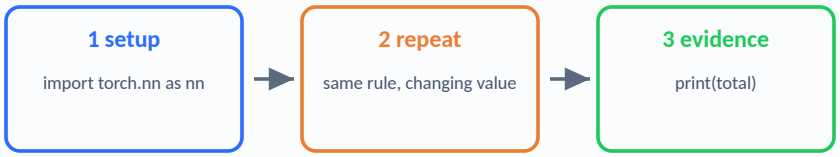
Reading target: count parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import torch.nn as nn` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	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(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (5->18->3 MLP): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: torch, nn, net, Sequential, Linear. Names are the reader's map of roles and responsibilities.

Layer	What to read
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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

🛠️ Reader to Writer Bridge

1. Read: explain count 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 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```python
import torch.nn as nn

net = nn.Sequential(nn.Linear(5, 18), nn.ReLU(), nn.Linear(18, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
165
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*

2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 184 Cosine Similarity (set 40)

💡 The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like cosine similarity into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*

- This project pushes the same idea into a less automatic situation.
- It builds on Project 183, where you practised count parameters, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 185 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

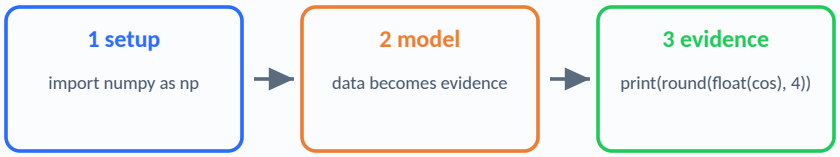
Skill step: pushes the same idea into a less automatic situation. It builds on Project 183: count parameters. Project 185 changes the surface problem while preserving the underlying method. *progress check*

Reading target: cosine 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 numpy as np` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(float(cos), 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, `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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: `queries, keys, values — attention computed by hand`.
2. Apply the rule	`a = np.array([5, 7, 2], dtype=float)`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something

Move	Code to inspect	Why it matters
		measurable. slow line
3. Check the evidence	`print(round(float(cos), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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 = np.array([5, 7, 2], dtype=float)`

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. 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	Cosine Similarity (set 40): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: numpy, np, array, dtype, cos. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([5, 7, 2], dtype=float) | Evidence: print(round(float(cos), 4))

Reader to Writer Bridge

1. Read: explain cosine 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 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(round(float(cos), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why 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 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Cosine similarity compares direction, ignoring magnitude — key for embeddings.

```
import numpy as np

a = np.array([5, 7, 2], dtype=float)
b = np.array([6, 7, 1], dtype=float)
cos = a @ b / (np.linalg.norm(a) * np.linalg.norm(b))
print(round(float(cos), 4))
```

```
output
0.989
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(cos), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained: choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 185 Sigmoid of -4.0

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like sigmoid of -4.0 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 184, where you practised cosine similarity, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 186 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

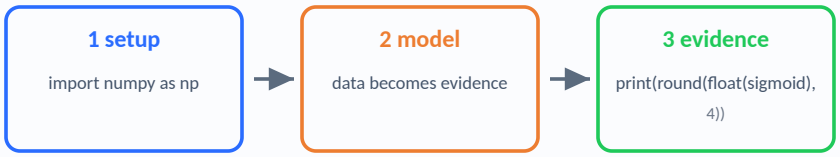
Skill step: pushes the same idea into a less automatic situation. It builds on Project 184: cosine similarity. Project 186 changes the surface problem while preserving the underlying method. *progress check*

Reading target: sigmoid of -4.0 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import numpy as np` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(float(sigmoid), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`sigmoid = 1 / (1 + np.exp(-z))`	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(round(float(sigmoid), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `sigmoid = 1 / (1 + np.exp(-z))`

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. 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	Sigmoid of -4.0: connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: numpy, np, sigmoid, exp, round. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: sigmoid = 1 / (1 + np.exp(-z)) | Evidence: print(round(float(sigmoid), 4))

🛠️ Reader to Writer Bridge

1. Read: explain sigmoid of -4.0 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 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(round(float(sigmoid), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of sigmoid of -4.0 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)

The sigmoid squashes any real number into (0, 1).

```
import numpy as np

z = -4.0
sigmoid = 1 / (1 + np.exp(-z))
print(round(float(sigmoid), 4))
```

```
output
0.018
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(sigmoid), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 186 Softmax of a 3-Vector (set 42)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like softmax of a 3-vector into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 185, where you practised sigmoid of -4.0, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 187 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

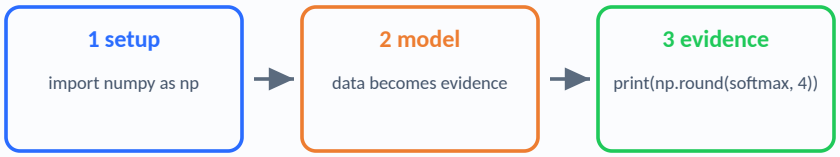
Skill step: pushes the same idea into a less automatic situation. It builds on Project 185: sigmoid of -4.0. Project 187 changes the surface problem while preserving the underlying method. *progress check*

Reading target: softmax of a 3-vector is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import numpy as np` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(np.round(softmax, 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`z = np.array([3.5, -3.0, -0.5])`	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(np.round(softmax, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `z = np.array([3.5, -3.0, -0.5])`

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. 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	Softmax of a 3-Vector (set 42): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.

Layer	What to read
Names	Watch these names first: numpy, np, array, exp, max. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: z = np.array([3.5, -3.0, -0.5]) | Evidence: print(np.round(softmax, 4))

Reader to Writer Bridge

1. Read: explain softmax of a 3-vector in one sentence before touching the 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(np.round(softmax, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of softmax of a 3-vector and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Softmax turns logits into a probability distribution.

```
import numpy as np

z = np.array([3.5, -3.0, -0.5])
e = np.exp(z - z.max())
softmax = e / e.sum()
print(np.round(softmax, 4))
```

```
output
[0.9806 0.0015 0.018 ]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(np.round(softmax, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 187 Dot Product (set 43)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like dot product into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 186, where you practised softmax of a 3-vector, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 188 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

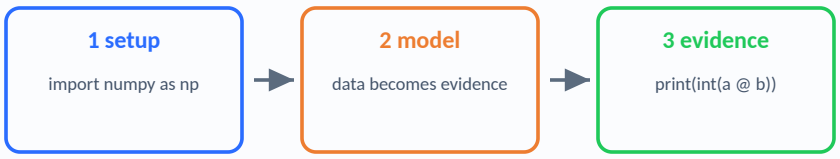
Skill step: pushes the same idea into a less automatic situation. It builds on Project 186: softmax of a 3-vector. Project 188 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(int(a @ 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.

Move	Code to inspect	Why it matters
2. Apply the rule	`a = np.array([7, 1, 4])`	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(int(a @ b))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax 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 = np.array([7, 1, 4])`

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. 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	Dot Product (set 43): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: numpy, np, array. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: a = np.array([7, 1, 4]) | Evidence: print(int(a @ b))

Reader to Writer Bridge

1. Read: explain dot product in one sentence before touching the 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(int(a @ b))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of dot product and name what must stay the 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 @ operator is matrix/vector multiplication.

```python
import numpy as np

a = np.array([7, 1, 4])
b = np.array([4, 9, 5])
print(int(a @ b))
```

```
output
57
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(int(a @ b))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.

- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 188 Binary Cross-Entropy (set 44)

The Idea Behind It

Under every ML library are a handful of vector operations. Implementing them in NumPy demystifies what the frameworks do. The deeper point is learning how Python turns a small intention like binary cross-entropy into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Transformers & Attention from Scratch, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 187, where you practised dot product, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

Skill Roadmap

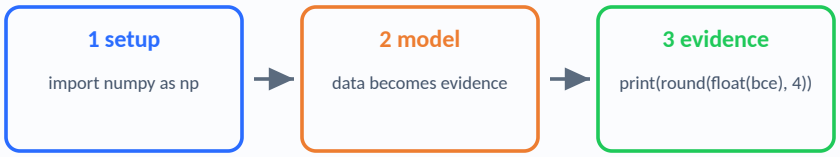
Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 187: dot product. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

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

Logic Lens

- Setup: `import numpy as np` 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: queries, keys, values — attention computed by hand.
- Evidence: `print(round(float(bce), 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, 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	`import numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: queries, keys, values — attention computed by hand.
2. Apply the rule	`y = np.array([1, 0, 1])`	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(round(float(bce), 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Transformers & Attention from Scratch becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `y = np.array([1, 0, 1])`

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. 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	Binary Cross-Entropy (set 44): connect the surface task to the chapter skill, queries, keys, values — attention computed by hand.
Names	Watch these names first: numpy, np, array, bce, log. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: y = np.array([1, 0, 1]) | Evidence: print(round(float(bce), 4))

🛠️ Reader to Writer Bridge

1. Read: explain binary cross-entropy in one sentence before touching the 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, 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(round(float(bce), 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of binary cross-entropy and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Cross-entropy penalises confident wrong predictions heavily.

```
import numpy as np

y = np.array([1, 0, 1])
p = np.array([0.7, 0.2, 0.5])
bce = -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()
print(round(float(bce), 4))
```

```
output
0.4243
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(float(bce), 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Vectorise it over a batch of inputs.

⚠ Common Mistakes

- Numerical overflow in exp — subtract the max first.
- Forgetting an epsilon inside log().

Skills gained choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Chapter Progress Checkpoint

What this chapter added: after 23 mini-projects, Transformers & Attention from Scratch should feel like a reusable Python move: queries, keys, values — attention computed by hand. *chapter checkpoint*

1 Read
Open any project from Projects 166-188 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 15 (LLM Engineering, Prompts & RAG) 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

Recall · Meaning · Structure · Transfer

14.1.Q1 Recall — Turn back to Project 166 (Multi-Head Split (d_model=128)). Without running it, what exactly does it print?

14.1.Q2 Meaning — Turn back to Project 168 (Attention Cost for seq_len=2048). Without running it, what exactly does it print?

14.1.Q3 Structure — Turn back to Project 169 (LLM Cost for 16000 Tokens). Without running it, what exactly does it print?

14.1.Q4 Transfer — What is the main problem this chapter's topic solves?

A. It gives a named, repeatable way to handle the chapter's programming situation.

B. It removes the need to test code.

C. It makes every program shorter.

D. It only matters for memorising syntax.

14.1.Q5 Recall What should you identify before running a project?

- A The setup, the rule or transformation, and the evidence line.
- B Only the final answer.
- C Only the imports.
- D Only the file name.

14.1.Q6 Meaning What is the most important edge case in Transformers & Attention from Scratch?

14.1.Q7 Structure What proves the skill is transferable?

Answer Key – Chapter 14 – Transformers & Attention from Scratch

14.1.Q1. 16

14.1.Q2. `4194304`

14.1.Q3. `0.008`

14.1.Q4. A — Attention scores how strongly one token should read information from other tokens.

14.1.Q5. A — prediction comes from setup, rule and evidence, not from guessing.

14.1.Q6. Self-attention cost grows with sequence length squared, so long context is expensive.

14.1.Q7. Add a value matrix and compute the weighted sum after softmax.

Chapter 15 · LLM Engineering, Prompts & RAG

Context windows, prompting, retrieval — engineering with language models.

Chapter Skill Map

What this chapter adds: In LLM Engineering, Prompts & RAG, the focus is context windows, prompting, retrieval — engineering with language models. You practise it across 10 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 15 assumes the reading habits from Chapter 14: Transformers & Attention from Scratch. You should carry forward the habit of predicting output before trusting the page.

Where it leads: Chapter 16, Scaling, Compute, Cost & MLOps, reuses this chapter's discipline with a new kind of problem.

1. **Projects 189-191**
   Recognise the pattern and read the code slowly. *name it first*
2. **Projects 192-194**
   Trace similar programs with less prompting and explain what changed.
3. **Projects 195-198**
   Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

LLM Engineering, Prompts & RAG: what problem does this solve for a working programmer?

name the move before typing

LLM engineering is context engineering: prompts, retrieval and constraints decide what the model can use.

This chapter's focus is: Context windows, prompting, retrieval — engineering with language models. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
docs = ["Java streams filter data", "RAG retrieves context"]
query = "retrieve context"
words = set(query.lower().split())
best = max(docs, key=lambda d: len(words & set(d.lower().split())))
print(best)
```

```
output
RAG retrieves context
```

The retriever chooses the document with the most overlapping query words. *predict first, verify second*

PROMPT

The instruction and context sent to a language model.

CONTEXT WINDOW

The token budget available for prompt plus response.

RAG

Retrieval-augmented generation: fetch relevant context before answering.

GROUNDING

Tying a model response to supplied evidence instead of unsupported guessing.

Beginner model

At beginner level, LLM Engineering, Prompts & RAG should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

Reliable LLM systems measure answer quality, citation support, latency and cost together.

The edge case to respect

Retrieval can fail silently if chunks, embeddings or ranking do not match the user's question.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Add a third document and explain which query terms make it win or lose.

Example 1

Trace the opening LLM Engineering, Prompts & RAG example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read LLM Engineering, Prompts & RAG as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Add a third document and explain which query terms make it win or lose.

⚠ Common Mistakes

- Copying the surface syntax for LLM Engineering, Prompts & RAG without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Retrieval can fail silently if chunks, embeddings or ranking do not match the user's question.
- Trusting the output before predicting it by hand.

This chapter contains 10 hands-on mini-projects (Projects 189–198). Type each one out, run it, and experiment.

Project 189 Call a Chat LLM

The Idea Behind It

Building with LLMs is its own engineering discipline: prompting, retrieval, tokens and grounding. The deeper point is learning how Python turns a small intention like call a chat llm into a readable sequence of `setup`, `logic`, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen `code reading`: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, `LLM Engineering, Prompts & RAG`, to a concrete mini-program.
- Read the program as `input or setup`, `transformation`, then `output`. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 190 will change the surface task again, so treat this one as a checkpoint: predict the

output before you look ahead. *predict first*

Skill Roadmap

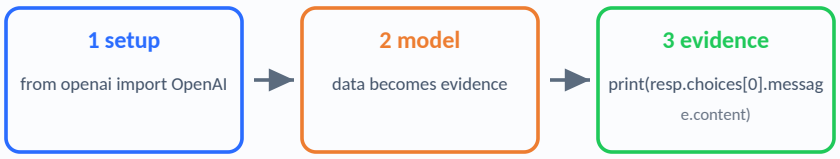
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 190 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `from openai import OpenAI` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(resp.choices[0].message.content)` exposes the result; match it 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	`from openai import OpenAI`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`model='gpt-4o-mini',`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(resp.choices[0].message.content)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `model='gpt-4o-mini',`

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

Code Reading Stack

Layer	What to read
Intent	Call a Chat LLM: connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: openai, OpenAI, client, resp, chat. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: from openai import OpenAI | Core: model='gpt-4o-mini', | Evidence: print(resp.choices[0].message.content)

Reader to Writer Bridge

1. Read: explain call a chat llm 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 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(resp.choices[0].message.content)`. Then compare punctuation, spacing, and order.

prediction before proof

2. Explain: point to `from openai import OpenAI` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of call a chat llm and name what must stay the 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 chat API takes a list of role/content messages and returns a reply.

```
from openai import OpenAI

client = OpenAI()
resp = client.chat.completions.create(
    model='gpt-4o-mini',
    messages=[{'role': 'user', 'content': 'Say hello in 3 words.'}])
print(resp.choices[0].message.content)
```

```
output
Hello there, friend!
```

How it works:

1. Start at `from openai import OpenAI`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(resp.choices[0].message.content)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Change k and observe the retrieved set.

⚠ Common Mistakes

- Stuffing more context than the window allows.
- Trusting ungrounded answers without retrieval.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 190 Build a RAG Prompt

The Idea Behind It

Building with LLMs is its own engineering discipline: prompting, retrieval, tokens and grounding. The deeper point is learning how Python turns a small intention like build a rag prompt into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 189, where you practised call a chat llm, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 191 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

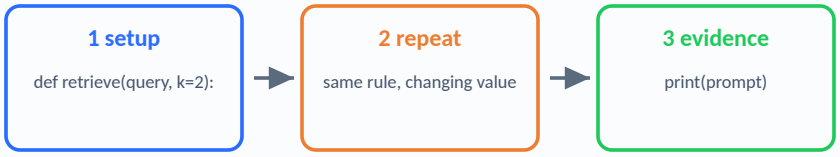
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 189: call a chat llm. Project 191 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `def retrieve(query, k=2):` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(prompt)` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 retrieve(query, k=2):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`"Embeddings map text to vectors for similarity search.",`	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(prompt)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `"Embeddings map text to vectors for similarity search.",`

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

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 RAG Prompt: connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: retrieve, query, corpus, retrieves, relevant. 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	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: def retrieve(query, k=2): | Core: "Embeddings map text to vectors for similarity search.", | Evidence: print(prompt)

🛠️ Reader to Writer Bridge

1. Read: explain build a rag prompt in one sentence before touching the 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, Project 191 should feel like a variation, not a brand-new mystery.

   from example to tool

✅ Project Self-Check

1. Predict: before running, write the exact output you expect from `print(prompt)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def retrieve(query, 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 build a rag prompt and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

retrieve() ranks corpus chunks by word overlap with the question; the prompt then instructs the model to answer only from that retrieved context.

```python
def retrieve(query, k=2):
    corpus = [
        "RAG retrieves relevant chunks before answering.",
        "Embeddings map text to vectors for similarity search.",
        "The context window limits how many chunks fit.",
    ]
    words = set(query.lower().split())
    ranked = sorted(corpus, key=lambda c: -len(words & set(c.lower().split())))
    return "\n".join(ranked[:k])

query = "How does RAG answer questions?"
context = retrieve(query, k=2)
prompt = f"""Answer using only the context.
Context:
{context}

Question: {query}"""
print(prompt)
```

```
output
Answer using only the context.
Context:
RAG retrieves relevant chunks before answering.
The context window limits how many chunks fit.

Question: How does RAG answer questions?
```

How it works:

1. Start at `def retrieve(query, k=2):`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(prompt)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change k and observe the retrieved set.

⚠ Common Mistakes

- Stuffing more context than the window allows.
- Trusting ungrounded answers without retrieval.

Skills gained tracing repeated work | reasoning through decisions | packaging logic into reusable steps | choosing and reading data structures

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 191 Vector Search (Top-k by Cosine)

The Idea Behind It

Building with LLMs is its own engineering discipline: prompting, retrieval, tokens and grounding. The deeper point is learning how Python turns a small intention like vector search into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.

- It builds on Project 190, where you practised build a rag prompt, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 192 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

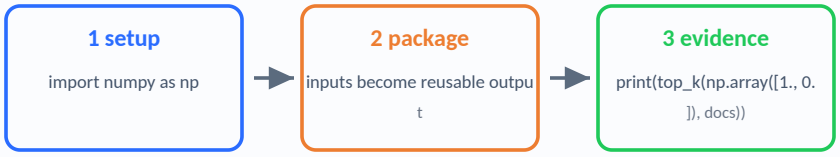
Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 190: build a rag prompt. Project 192 changes the surface problem while preserving the underlying method. *progress check*

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

🔎 Logic Lens

- Setup: `import numpy as np` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(top_k(np.array([1., 0.]), docs))` exposes the result; match it against the output before changing the program. *output proves the trace*

🖼️ Visual Model



Read the project through this picture first: setup creates the facts, package applies the 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 numpy as np`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`return np.argsort(-sims)[:k]`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something

Move	Code to inspect	Why it matters
		measurable. slow line
3. Check the evidence	`print(top_k(np.array([1., 0.]), docs))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax 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 np.argsort(-sims)[:k]`

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.

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	Vector Search (Top-k by Cosine): connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: numpy, np, top_k, query, docs. Names are the reader's map of roles and responsibilities.
Flow	imports load outside tools; functions/methods create reusable boundaries.
Data shape	Vector or array data: one expression may transform many values at once.
Evidence	Entry: import numpy as np | Core: return np.argsort(-sims)[:k] | Evidence: print(top_k(np.array([1., 0.]), docs))

Reader to Writer Bridge

1. Read: explain vector search in one sentence before touching the code; the goal gives the syntax context.
2. Trace: follow the actual values from setup to output, then carry forward the move from Project 190.

   read with a pencil
3. Change: alter one value, condition, loop bound, object, or label and predict the exact output.

4. Create: 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(top_k(np.array([1., 0.]), docs))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import numpy as np` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of vector search and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Retrieval ranks documents by embedding similarity to the query.

```
import numpy as np

def top_k(query, docs, k=2):
    sims = docs @ query / (np.linalg.norm(docs, axis=1) *
np.linalg.norm(query))
    return np.argsort(-sims)[:k]

docs = np.array([[1., 0.], [0.9, 0.1], [0., 1.]])
print(top_k(np.array([1., 0.]), docs))
```

```
output
[0 1]
```

How it works:

1. Start at `import numpy as np`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(top_k(np.array([1., 0.]), docs))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Change k and observe the retrieved set.

⚠ Common Mistakes

- Stuffing more context than the window allows.
- Trusting ungrounded answers without retrieval.

Skills gained packaging logic into reusable steps | choosing and reading data structures | recognising algorithm patterns | connecting code to AI engineering concepts

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 192 Count Tokens for an LLM

The Idea Behind It

Building with LLMs is its own engineering discipline: prompting, retrieval, tokens and grounding. The deeper point is learning how Python turns a small intention like count tokens for an llm into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 191, where you practised vector search, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 193 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

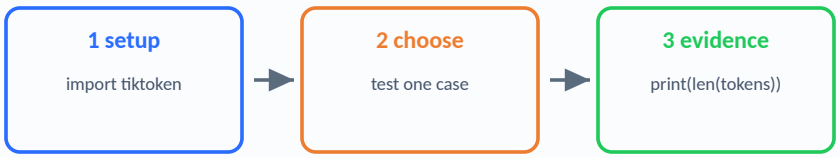
Skill step: moves you from recognition toward independent tracing. It builds on Project 191: vector search. Project 193 changes the surface problem while preserving the underlying method. *progress check*

Reading target: count tokens for an llm is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

🔎 Logic Lens

- Setup: `import tiktoken` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(len(tokens))` exposes the result; match it 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 tiktoken`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`enc = tiktoken.get_encoding('cl100k_base')`	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(len(tokens))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `enc = tiktoken.get_encoding('cl100k_base')`

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. 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

Choose one value that takes the other branch and explain why the test changes.

Code Reading Stack

Layer	What to read
Intent	Count Tokens for an LLM: connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: tiktoken, enc, get_encoding, cl100k_base, tokens. 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 tiktoken | Core: enc = tiktoken.get_encoding('cl100k_base') | Evidence: print(len(tokens))

🛠️ Reader to Writer Bridge

1. Read: explain count tokens for an llm in one sentence before touching the 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(len(tokens))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import tiktoken` and say why 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 tokens for an llm and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Billing and context limits are measured in tokens, not characters.

```
import tiktoken

enc = tiktoken.get_encoding('cl100k_base')
tokens = enc.encode('How many tokens is this?')
print(len(tokens))
```

```
output
6
```

How it works:

1. Start at `import tiktoken`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(len(tokens))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change k and observe the retrieved set.

⚠ Common Mistakes

- Stuffing more context than the window allows.
- Trusting ungrounded answers without retrieval.

Skills gained reasoning through decisions | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 193 Chunk a Document for Indexing

The Idea Behind It

Building with LLMs is its own engineering discipline: prompting, retrieval, tokens and grounding. The deeper point is learning how Python turns a small intention like chunk a document for indexing into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 192, where you practised count tokens for an llm, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 194 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 192: count tokens for an llm. Project 194 changes the surface problem while preserving the underlying method. *progress check*

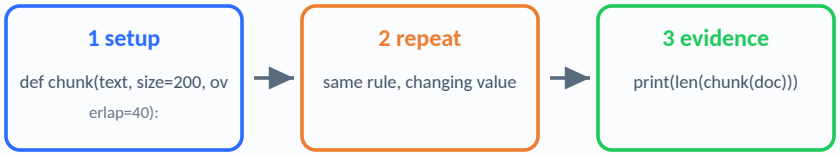
Reading target: chunk a document for indexing is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `def chunk(text, size=200, overlap=40):` 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: context windows, prompting, retrieval — engineering with language models.

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

Visual Model



Read the project through this picture first: setup creates the facts, repeat applies the 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 chunk(text, size=200, overlap=40):`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`return [text[i:i+size] for i in range(0, len(text), step)]`	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(len(chunk(doc)))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax 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 [text[i:i+size] for i in range(0, len(text), step)]`

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

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

Code Reading Stack

Layer	What to read
Intent	Chunk a Document for Indexing: connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: chunk, text, size, overlap, step. Names are the reader's map of roles and responsibilities.
Flow	functions/methods create reusable boundaries; loops repeat a rule.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: def chunk(text, size=200, overlap=40): | Core: return [text[i:i+size] for i in range(0, len(text), step)] | Evidence: print(len(chunk(doc)))

Reader to Writer Bridge

1. Read: explain chunk a document for indexing in one sentence before touching the 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(len(chunk(doc)))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `def chunk(text, size=200, overlap=40):` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of chunk a document for indexing and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Overlapping chunks keep context from being split across boundaries.

```python
def chunk(text, size=200, overlap=40):
    step = size - overlap
    return [text[i:i+size] for i in range(0, len(text), step)]

doc = 'x' * 500
print(len(chunk(doc)))
```

```
output
4
```

How it works:

1. Start at `def chunk(text, size=200, overlap=40):`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(len(chunk(doc)))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Change k and observe the retrieved set.

⚠ Common Mistakes

- Stuffing more context than the window allows.
- Trusting ungrounded answers without retrieval.

Skills gained tracing repeated work | packaging logic into reusable steps | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 194 3 SGD Steps (lr=0.07)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like 3 sgd steps into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 193, where you practised chunk a document for indexing, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 195 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

🧭 Skill Roadmap

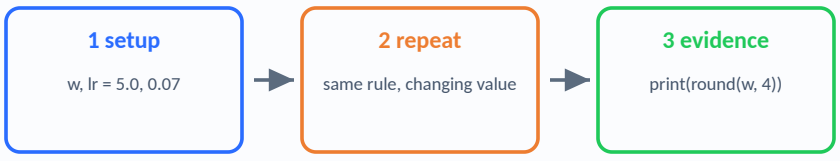
Skill step: moves you from recognition toward independent tracing. It builds on Project 193: chunk a document for indexing. Project 195 changes the surface problem while preserving the underlying method. *progress check*

Reading target: 3 sgd steps is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `w, lr = 5.0, 0.07` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(round(w, 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	`w, lr = 5.0, 0.07`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`for _ in range(3):`	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(round(w, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax 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 _ in range(3):`

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. Colon: Opens Python blocks, separates keys from values, or 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	3 SGD Steps (lr=0.07): connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: lr, range, grad, round. Names are the reader's map of roles and responsibilities.
Flow	loops repeat a rule.
Data shape	Model pipeline data: features, labels, tensors, prompts, embeddings, predictions, or metrics.
Evidence	Entry: w, lr = 5.0, 0.07 | Core: for _ in range(3): | Evidence: print(round(w, 4))

Reader to Writer Bridge

1. Read: explain 3 sgd 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 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(round(w, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `w, lr = 5.0, 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 3 sgd 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)

Smaller learning rates converge more slowly but more stably.

```
w, lr = 5.0, 0.07
for _ in range(3):
    grad = 2 * (w - 1)
    w -= lr * grad
print(round(w, 4))
```

```
output
3.5442
```

How it works:

1. Start at `w, lr = 5.0, 0.07`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(w, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 195 Attention Cost for seq_len=256

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=256 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 194, where you practised 3 sgd steps, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 196 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

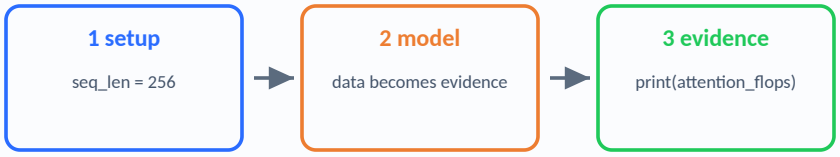
Skill step: pushes the same idea into a less automatic situation. It builds on Project 194: 3 sgd steps. Project 196 changes the surface problem while preserving the underlying method. *progress check*

Reading target: attention cost for seq_len=256 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `seq_len = 256` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 256`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the

behaviour.

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=256: connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. 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: seq_len = 256 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

🛠️ Reader to Writer Bridge

1. Read: explain attention cost for seq_len=256 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 256` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=256 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 256
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
65536
```

How it works:

1. Start at `seq_len = 256`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 196 LLM Cost for 2000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 2000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 195, where you practised attention cost for seq_len=256, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 197 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 195: attention cost for seq_len=256. Project 197 changes the surface problem while preserving the underlying method.

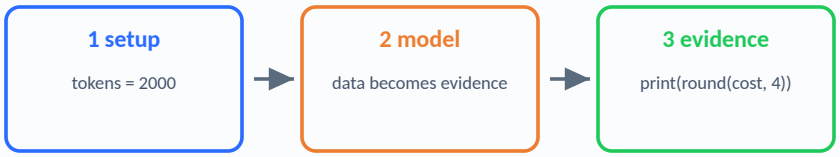
progress check

Reading target: llm cost for 2000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `tokens = 2000` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(round(cost, 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, 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	`tokens = 2000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	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(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 2000 Tokens: connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.

Layer	What to read
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 2000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

🛠️ Reader to Writer Bridge

1. Read: explain llm cost for 2000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 2000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 2000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 2000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.001
```

How it works:

1. Start at `tokens = 2000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*

2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 197 Multi-Head Split (d_model=24)

💡 The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*

- This project pushes the same idea into a less automatic situation.
- It builds on Project 196, where you practised llm cost for 2000 tokens, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 198 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

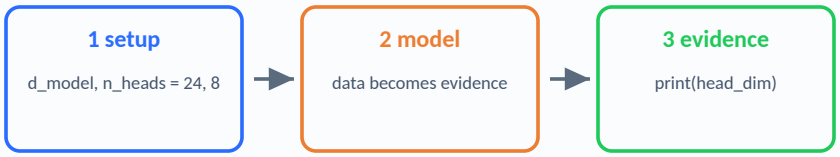
Skill step: pushes the same idea into a less automatic situation. It builds on Project 196: llm cost for 2000 tokens. Project 198 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `d_model, n_heads = 24, 8` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 24, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.

Move	Code to inspect	Why it matters
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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

Code Reading Stack

Layer	What to read
Intent	Multi-Head Split (d_model=24): connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. 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: d_model, n_heads = 24, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 24, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```
d_model, n_heads = 24, 8
assert d_model % n_heads == 0, 'must divide evenly'
head_dim = d_model // n_heads
print(head_dim)
```

```
output
3
```

How it works:

1. Start at `d_model, n_heads = 24, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.

- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 198 Count Parameters (6->16->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, LLM Engineering, Prompts & RAG, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project closes the chapter by asking you to apply the pattern with less hand-holding.
- It builds on Project 197, where you practised multi-head split, and asks you to reuse the same reasoning pattern. reuse the pattern After this, you should be able to explain the chapter pattern without leaning on the exact values used here. own the pattern

Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 197: multi-head split. By the end of this chapter, you should be able to explain the pattern without copying the exact example. progress check

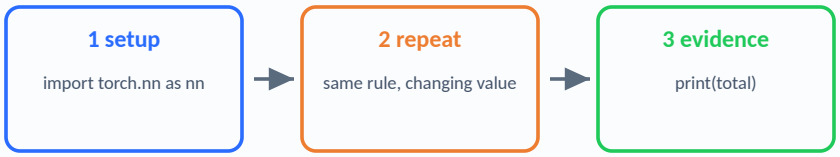
Reading target: count parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `import torch.nn as nn` 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: context windows, prompting, retrieval — engineering with language models.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: context windows, prompting, retrieval — engineering with language models.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	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(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: LLM Engineering, Prompts & RAG becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (6->16->3 MLP): connect the surface task to the chapter skill, context windows, prompting, retrieval — engineering with language models.
Names	Watch these names first: torch, nn, net, Sequential, Linear. 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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

🛠️ Reader to Writer Bridge

1. Read: explain count 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 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, 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```
import torch.nn as nn

net = nn.Sequential(nn.Linear(6, 16), nn.ReLU(), nn.Linear(16, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
163
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 10 mini-projects, LLM Engineering, Prompts & RAG should feel like a reusable Python move: context windows, prompting, retrieval — engineering with language models.

chapter checkpoint

1 Read
Open any project from Projects 189-198 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 16 (Scaling, Compute, Cost & MLOps) 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

Recall · Meaning · Structure · Transfer

15.1.Q1 Recall — Turn back to Project 193 (Chunk a Document for Indexing). Without running it, what exactly does it print?

15.1.Q2 Meaning — Turn back to Project 194 (3 SGD Steps (lr=0.07)). Without running it, what exactly does it print?

15.1.Q3 Structure — Turn back to Project 195 (Attention Cost for seq_len=256). Without running it, what exactly does it print?

15.1.Q4 Transfer — What is the main problem this chapter's topic solves?

- A. It gives a named, repeatable way to handle the chapter's programming situation.
- B. It removes the need to test code.
- C. It makes every program shorter.
- D. It only matters for memorising syntax.

15.1.Q5 Recall — What should you identify before running a project?

- A. The setup, the rule or transformation, and the evidence line.
- B. Only the final answer.
- C. Only the imports.
- D. Only the file name.

15.1.Q6 Meaning What is the most important edge case in LLM Engineering, Prompts & RAG?

15.1.Q7 Structure What proves the skill is transferable?

Answer Key – Chapter 15 – LLM Engineering, Prompts & RAG

15.1.Q1. 4

15.1.Q2. 3.5442

15.1.Q3. 65536

15.1.Q4. A — LLM engineering is context engineering: prompts, retrieval and constraints decide what the model can use.

15.1.Q5. A — prediction comes from setup, rule and evidence, not from guessing.

15.1.Q6. Retrieval can fail silently if chunks, embeddings or ranking do not match the user's question.

15.1.Q7. Add a third document and explain which query terms make it win or lose.

Chapter 16 · Scaling, Compute, Cost & MLOps

Parameter counts, FLOPs, token economics and serving.

Chapter Skill Map

What this chapter adds: In Scaling, Compute, Cost & MLOps, the focus is parameter counts, flops, token economics and serving. You practise it across 11 small Python decisions, building a repeatable reading-and-writing habit rather than racing through examples. *chapter arc*

What it builds on: Chapter 16 assumes the reading habits from Chapter 15: LLM Engineering, Prompts & RAG. 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 199-201**
   Recognise the pattern and read the code slowly. *name it first*
2. **Projects 202-204**
   Trace similar programs with less prompting and explain what changed.
3. **Projects 205-209**
   Modify the idea, predict the result, then verify with output. *own the move*

The Idea Behind It

Scaling, Compute, Cost & MLOps: what problem does this solve for a working programmer?

name the move before typing

Scaling and MLOps make model work operational: count parameters, estimate cost, serve, monitor and roll back.

This chapter's focus is: Parameter counts, FLOPs, token economics and serving. Read every project in three passes: setup, rule, evidence. First find the values or data shape, then name the operation, then check the line that proves the result.

```
params = 4 * 8 + 8 + 8 * 3 + 3
print(params)
print(f"{params * 4 / 1024:.2f} KB")
```

```
output
67
0.26 KB
```

A Linear(4,8) plus Linear(8,3) model has 67 learned numbers. *predict first, verify second*

PARAMETER COUNT

The number of learned weights and biases in a model.

FLOPS

Floating-point operations, a rough measure of compute cost.

SERVING

Running a model for real requests under latency and reliability constraints.

DRIFT

A change in real-world data that makes old model behavior less reliable.

Beginner model

At beginner level, Scaling, Compute, Cost & MLOps should feel concrete: you can say what the program starts with, what one rule it applies, and what evidence proves it worked. Do not rush past the small example; it is the handle for the whole chapter.

What changes as the problems get harder

MLOps treats a model as a living system with versions, monitoring, budgets and rollback plans.

The edge case to respect

Weight memory is only the start; training also needs activations, gradients and optimizer state.

How to study the project sequence

The projects in this chapter are not random drills. They vary one surface detail at a time so you can separate the stable idea from the changing values. Your job is to predict, run, explain and then transfer: Double the hidden width and recount the parameters before changing code.

Example 1

Trace the opening Scaling, Compute, Cost & MLOps example by hand.

SOLUTION

1. Name the starting values and their types before reading the operation.
2. Point to the one line that applies the chapter idea.
3. Compare the exact output with your prediction, including punctuation and spacing.

A correct trace names the data, the rule and the evidence before the program is trusted.

Example 2

Turn the example into a sibling AI engineering project.

SOLUTION

1. Keep the same skeleton but change one input, rule or data shape.
2. Write the expected output before running.
3. If the output differs, explain whether the bug is in the code or in your mental trace.

Transfer proves you understand the idea rather than the exact values.

Mental-Math Shortcut

Read Scaling, Compute, Cost & MLOps as a repeatable loop: identify the setup, name the rule, predict the evidence, then change one detail. Double the hidden width and recount the parameters before changing code.

⚠ Common Mistakes

- Copying the surface syntax for Scaling, Compute, Cost & MLOps without naming the underlying rule.
- Changing several details at once, then being unable to explain which change caused the output.
- Weight memory is only the start; training also needs activations, gradients and optimizer state.
- Trusting the output before predicting it by hand.

This chapter contains 11 hands-on mini-projects (Projects 199–209). Type each one out, run it, and experiment.

Project 199 LLM Cost for 30000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 30000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project opens the chapter by setting the first concrete pattern.
- It starts this chapter's project sequence by turning the chapter focus into a small runnable example.

first anchor Project 200 will change the surface task again, so treat this one as a checkpoint: predict the

output before you look ahead. *predict first*

Skill Roadmap

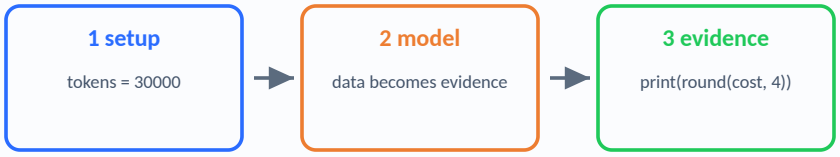
Skill step: opens the chapter by setting the first concrete pattern. It starts a new chapter sequence, so the first job is to name the pattern before typing it. Project 200 changes the surface problem while preserving the underlying method. *progress check*

Reading target: llm cost for 30000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `tokens = 30000` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(round(cost, 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, 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	`tokens = 30000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something measurable. *slow line*

Move	Code to inspect	Why it matters
3. Check the evidence	`print(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 30000 Tokens: connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 30000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

Reader to Writer Bridge

1. Read: explain llm cost for 30000 tokens 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 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 30000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 30000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 30000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.015
```

How it works:

1. Start at `tokens = 30000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 200 Multi-Head Split (d_model=248)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 199, where you practised llm cost for 30000 tokens, and asks you to reuse the same reasoning pattern. reuse the pattern Project 201 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 199: llm cost for 30000 tokens. Project 201 changes the surface problem while preserving the underlying method.

progress check

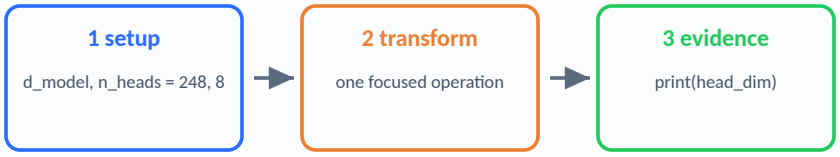
Reading target: multi-head split is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `d_model, n_heads = 248, 8` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 248, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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. 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	Multi-Head Split (d_model=248): connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. 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: d_model, n_heads = 248, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 248, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```python
d_model, n_heads = 248, 8
assert d_model % n_heads == 0, 'must divide evenly'
head_dim = d_model // n_heads
print(head_dim)
```

```
output
31
```

How it works:

1. Start at `d_model, n_heads = 248, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 201 Count Parameters (34->8->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project adds one small variation while the chapter pattern is still new.
- It builds on Project 200, where you practised multi-head split, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 202 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: adds one small variation while the chapter pattern is still new. It builds on Project 200: multi-head split. Project 202 changes the surface problem while preserving the underlying method. *progress check*

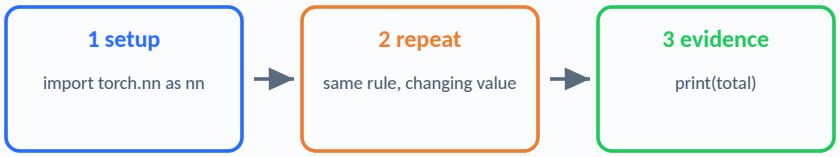
Reading target: count parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `import torch.nn as nn` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	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(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (34->8->3 MLP): connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: torch, nn, net, Sequential, Linear. 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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

Reader to Writer Bridge

1. Read: explain count 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 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)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```
import torch.nn as nn

net = nn.Sequential(nn.Linear(34, 8), nn.ReLU(), nn.Linear(8, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
307
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.

4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 202 Attention Cost for seq_len=3968

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=3968 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 201, where you practised count parameters, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 203 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 201: count parameters. Project 203 changes the surface problem while preserving the underlying method.

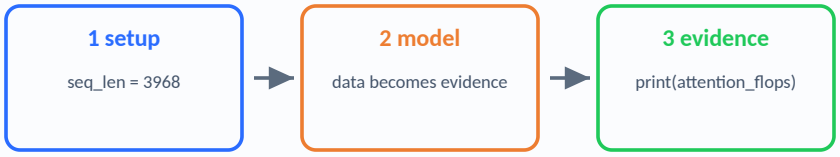
progress check

Reading target: attention cost for seq_len=3968 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `seq_len = 3968` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 3968`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=3968: connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. 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: seq_len = 3968 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

🛠️ Reader to Writer Bridge

1. Read: explain attention cost for seq_len=3968 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 3968` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=3968 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 3968
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
15745024
```

How it works:

1. Start at `seq_len = 3968`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 203 LLM Cost for 31000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 31000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 202, where you practised attention cost for seq_len=3968, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 204 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 202: attention cost for seq_len=3968. Project 204 changes the surface problem while preserving the underlying method.

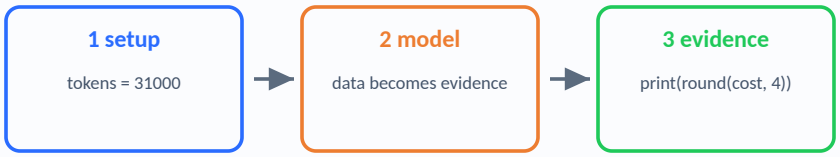
progress check

Reading target: llm cost for 31000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `tokens = 31000` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(round(cost, 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, 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	`tokens = 31000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	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(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 31000 Tokens: connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: tokens, price_per_1k, cost, round. Names are the reader's map of roles and responsibilities.

Layer	What to read
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: tokens = 31000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

🛠️ Reader to Writer Bridge

1. Read: explain llm cost for 31000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 31000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 31000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 31000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.0155
```

How it works:

1. Start at `tokens = 31000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 204 Multi-Head Split (d_model=256)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.

- It builds on Project 203, where you practised llm cost for 31000 tokens, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 205 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

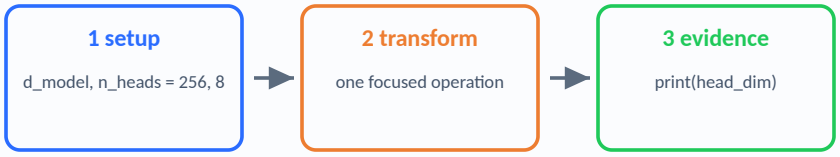
Skill step: moves you from recognition toward independent tracing. It builds on Project 203: llm cost for 31000 tokens. Project 205 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `d_model, n_heads = 256, 8` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 256, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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. 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	Multi-Head Split (d_model=256): connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. 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: d_model, n_heads = 256, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*

2. Explain: point to `d_model, n_heads = 256, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```
d_model, n_heads = 256, 8
assert d_model % n_heads == 0, 'must divide evenly'
head_dim = d_model // n_heads
print(head_dim)
```

```
output
32
```

How it works:

1. Start at `d_model, n_heads = 256, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 205 3 SGD Steps (lr=0.12)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like 3 sgd steps into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project moves you from recognition toward independent tracing.
- It builds on Project 204, where you practised multi-head split, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 206 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

Skill step: moves you from recognition toward independent tracing. It builds on Project 204: multi-head split. Project 206 changes the surface problem while preserving the underlying method. *progress check*

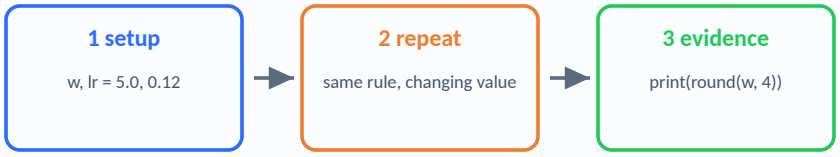
Reading target: 3 sgd steps is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `w, lr = 5.0, 0.12` 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: parameter counts, flops, token economics and serving.

- Evidence: `print(round(w, 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	`w, lr = 5.0, 0.12`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`for _ in range(3):`	This line controls repeated work. Name what changes each pass and what stays fixed. *slow line*
3. Check the evidence	`print(round(w, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax 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 _ in range(3):`

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.

Build-it-yourself move

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

Rewrite drill

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

Code Reading Stack

Layer	What to read
Intent	3 SGD Steps (lr=0.12): connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: lr, range, grad, 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: w, lr = 5.0, 0.12 | Core: for _ in range(3): | Evidence: print(round(w, 4))

🛠️ Reader to Writer Bridge

1. Read: explain 3 sgd 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 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(round(w, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `w, lr = 5.0, 0.12` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of 3 sgd 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)

Smaller learning rates converge more slowly but more stably.

```python
w, lr = 5.0, 0.12
for _ in range(3):
    grad = 2 * (w - 1)
    w -= lr * grad
print(round(w, 4))
```

```
output
2.7559
```

How it works:

1. Start at `w, lr = 5.0, 0.12`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(w, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 206 Attention Cost for seq_len=384

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like attention cost for seq_len=384 into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 205, where you practised 3 sgd steps, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 207 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

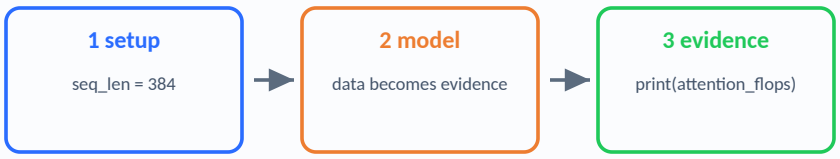
Skill step: pushes the same idea into a less automatic situation. It builds on Project 205: 3 sgd steps. Project 207 changes the surface problem while preserving the underlying method. *progress check*

Reading target: attention cost for seq_len=384 is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

Logic Lens

- Setup: `seq_len = 384` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(attention_flops)` exposes the result; match it 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	`seq_len = 384`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.

Move	Code to inspect	Why it matters
2. Apply the rule	`attention_flops = seq_len ** 2 # pairwise scores`	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(attention_flops)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `attention_flops = seq_len ** 2 # pairwise scores`

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.

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	Attention Cost for seq_len=384: connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: seq_len, attention_flops, pairwise, scores. 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: seq_len = 384 | Core: attention_flops = seq_len ** 2 # pairwise scores | Evidence: print(attention_flops)

Reader to Writer Bridge

1. Read: explain attention cost for seq_len=384 in one sentence before touching the 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(attention_flops)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `seq_len = 384` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of attention cost for seq_len=384 and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Self-attention scales quadratically with sequence length.

```
seq_len = 384
attention_flops = seq_len ** 2  # pairwise scores
print(attention_flops)
```

```
output
147456
```

How it works:

1. Start at `seq_len = 384`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(attention_flops)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.

- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. same map, bigger code

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. tiny program, full habit

Project 207 LLM Cost for 3000 Tokens

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like llm cost for 3000 tokens into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. why this line now?

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. eyes before keyboard

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. three-pass reading
- This project pushes the same idea into a less automatic situation.
- It builds on Project 206, where you practised attention cost for seq_len=384, and asks you to reuse the same reasoning pattern. reuse the pattern Project 208 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. predict first

Skill Roadmap

Skill step: pushes the same idea into a less automatic situation. It builds on Project 206: attention cost for seq_len=384. Project 208 changes the surface problem while preserving the underlying method.

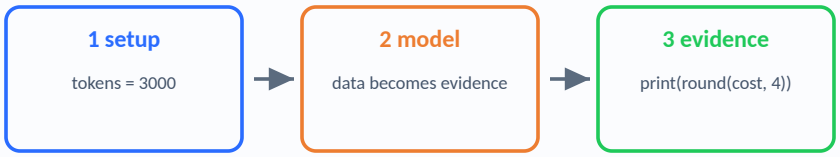
progress check

Reading target: llm cost for 3000 tokens is the surface task; the transferable Python move is to mark setup, rule, and output before running. setup -> rule -> result

Logic Lens

- Setup: `tokens = 3000` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(round(cost, 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, 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	`tokens = 3000`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`cost = tokens / 1000 * price_per_1k`	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(round(cost, 4))`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. *proof line*

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `cost = tokens / 1000 * price_per_1k`

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. 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	LLM Cost for 3000 Tokens: connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: tokens, price_per_1k, cost, round. 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: tokens = 3000 | Core: cost = tokens / 1000 * price_per_1k | Evidence: print(round(cost, 4))

🛠️ Reader to Writer Bridge

1. Read: explain llm cost for 3000 tokens in one sentence before touching the 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(round(cost, 4))`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `tokens = 3000` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of llm cost for 3000 tokens and name what must stay the same: the data shape, the rule, the branch, the loop pattern, or the output format. *same idea, new surface*

🤖 From Concept to Code (Python)

Token-based pricing makes prompt length a real budget line.

```
tokens = 3000
price_per_1k = 0.0005
cost = tokens / 1000 * price_per_1k
print(round(cost, 4))
```

```
output
0.0015
```

How it works:

1. Start at `tokens = 3000`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(round(cost, 4))`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ Where You'll Use This: You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 208 Multi-Head Split (d_model=32)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like multi-head split into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project pushes the same idea into a less automatic situation.
- It builds on Project 207, where you practised llm cost for 3000 tokens, and asks you to reuse the same reasoning pattern. *reuse the pattern* Project 209 will change the surface task again, so treat this one as a checkpoint: predict the output before you look ahead. *predict first*

Skill Roadmap

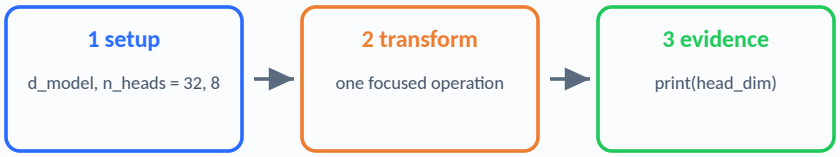
Skill step: pushes the same idea into a less automatic situation. It builds on Project 207: llm cost for 3000 tokens. Project 209 changes the surface problem while preserving the underlying method. *progress check*

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

Logic Lens

- Setup: `d_model, n_heads = 32, 8` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(head_dim)` exposes the result; match it 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	`d_model, n_heads = 32, 8`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`assert d_model % n_heads == 0, 'must divide evenly'`	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(head_dim)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax shape yourself, not just recognise it on the page. Slow down at the core line and ask which token creates the behaviour.

CORE LINE `assert d_model % n_heads == 0, 'must divide evenly'`

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. 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	Multi-Head Split (d_model=32): connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: d_model, n_heads, must, divide, evenly. Names are the reader's map of roles and responsibilities.
Flow	Straight-line flow: read top to bottom as setup, operation, evidence.

Layer	What to read
Data shape	Scalar or small structured data: read names, literal values, and the operation that connects them.
Evidence	Entry: d_model, n_heads = 32, 8 | Core: assert d_model % n_heads == 0, 'must divide evenly' | Evidence: print(head_dim)

🛠️ Reader to Writer Bridge

1. Read: explain multi-head split in one sentence before touching the 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(head_dim)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `d_model, n_heads = 32, 8` and say why this line, value, import, or entry point is needed before the rest of the Python can work.
3. Transfer: change one small part of multi-head split and name what must stay the 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 attention head works in a d_model / n_heads subspace.

```python
d_model, n_heads = 32, 8
assert d_model % n_heads == 0, 'must divide evenly'
head_dim = d_model // n_heads
print(head_dim)
```

```
output
4
```

How it works:

1. Start at `d_model, n_heads = 32, 8`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.

3. Finish at `print(head_dim)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

Project 209 Count Parameters (7->14->3 MLP)

The Idea Behind It

Engineering LLMs and deep nets means reasoning about parameters, compute, learning rates and cost. The deeper point is learning how Python turns a small intention like count parameters into a readable sequence of setup, logic, and visible result. Treat the literal values as test cases; the durable lesson is why each line appears in this order. *why this line now?*

What You'll Build & Learn

Use this project to strengthen code reading: identify the starting data, follow the operation, and check the exact output against your prediction. *eyes before keyboard*

- Connect the chapter idea, Scaling, Compute, Cost & MLOps, to a concrete mini-program.
- Read the program as input or setup, transformation, then output. *three-pass reading*
- This project closes the chapter by asking you to apply the pattern with less hand-holding.

- It builds on Project 208, where you practised multi-head split, and asks you to reuse the same reasoning pattern. *reuse the pattern* After this, you should be able to explain the chapter pattern without leaning on the exact values used here. *own the pattern*

🧭 Skill Roadmap

Skill step: closes the chapter by asking you to apply the pattern with less hand-holding. It builds on Project 208: multi-head split. By the end of this chapter, you should be able to explain the pattern without copying the exact example. *progress check*

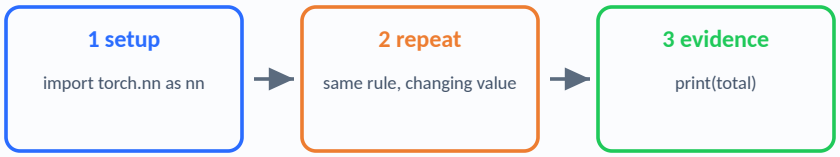
Reading target: count parameters is the surface task; the transferable Python move is to mark setup, rule, and output before running. *setup -> rule -> result*

🔎 Logic Lens

- Setup: `import torch.nn as nn` 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: parameter counts, flops, token economics and serving.
- Evidence: `print(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 torch.nn as nn`	This gives the project its starting fact, tool, object, or entry point. It connects the visible code to the chapter focus: parameter counts, flops, token economics and serving.
2. Apply the rule	`total = sum(p.numel() for p in net.parameters())`	This is where raw data becomes evidence: a model, metric, tensor, or pipeline step turns inputs into something

Move	Code to inspect	Why it matters
		measurable. slow line
3. Check the evidence	`print(total)`	The result becomes visible or reusable here. Confirm that your trace explains the exact output. proof line

Syntax Construction Lab

Construction focus: Scaling, Compute, Cost & MLOps becomes practical when you can build the syntax 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(p.numel() for p in net.parameters())`

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. 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 Parameters (7->14->3 MLP): connect the surface task to the chapter skill, parameter counts, flops, token economics and serving.
Names	Watch these names first: torch, nn, net, Sequential, Linear. 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 torch.nn as nn | Core: total = sum(p.numel() for p in net.parameters()) | Evidence: print(total)

Reader to Writer Bridge

1. Read: explain count 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 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, 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(total)`. Then compare punctuation, spacing, and order. *prediction before proof*
2. Explain: point to `import torch.nn as nn` and say why 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 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)

Each Linear layer has in×out weights plus one bias per output.

```python
import torch.nn as nn

net = nn.Sequential(nn.Linear(7, 14), nn.ReLU(), nn.Linear(14, 3))
total = sum(p.numel() for p in net.parameters())
print(total)
```

```
output
157
```

How it works:

1. Start at `import torch.nn as nn`: this establishes the first value, structure, or entry point the rest of the program depends on. *anchor line*
2. Move through the middle lines and name the one operation that changes information into a result; that is the logic you are practising, not the decorative variable names.
3. Finish at `print(total)`: the program makes its answer visible, so compare punctuation, spacing, and order exactly with the expected output. *evidence line*

Try it:

1. Change one input value or literal and predict the new output before running it.
2. Rename the variables or labels while keeping the logic identical, then confirm the behaviour did not change.
3. Add one extra output line that exposes an intermediate value so you can see the logic working.
4. Scale the numbers up and watch the cost grow.

⚠ Common Mistakes

- d_model not divisible by the number of heads.
- Ignoring the quadratic cost of long contexts.

Skills gained tracing repeated work | connecting code to AI engineering concepts | predicting exact output

→ **Where You'll Use This:** You will reuse this habit whenever you read unfamiliar Python: locate the data, name the transformation, and verify the output. *same map, bigger code*

⚡ Mental Model

Read the project as a three-step pipeline: set up the facts -> apply one focused rule -> make the result observable. *tiny program, full habit*

✅ Chapter Progress Checkpoint

What this chapter added: after 11 mini-projects, Scaling, Compute, Cost & MLOps should feel like a reusable Python move: parameter counts, flops, token economics and serving. *chapter checkpoint*

1 Read
Open any project from Projects 199-209 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

Recall · Meaning · Structure · Transfer

16.1.Q1 Recall — Turn back to Project 199 (LLM Cost for 30000 Tokens). Without running it, what exactly does it print?

16.1.Q2 Meaning — Turn back to Project 200 (Multi-Head Split (d_model=248)). Without running it, what exactly does it print?

16.1.Q3 Structure — Turn back to Project 202 (Attention Cost for seq_len=3968). Without running it, what exactly does it print?

16.1.Q4 Transfer What is the main problem this chapter's topic solves?

- A It gives a named, repeatable way to handle the chapter's programming situation.
- B It removes the need to test code.
- C It makes every program shorter.
- D It only matters for memorising syntax.

16.1.Q5 Recall What should you identify before running a project?

- A The setup, the rule or transformation, and the evidence line.
- B Only the final answer.
- C Only the imports.
- D Only the file name.

16.1.Q6 Meaning What is the most important edge case in Scaling, Compute, Cost & MLOps?

16.1.Q7 Structure What proves the skill is transferable?

Answer Key – Chapter 16 – Scaling, Compute, Cost & MLOps

16.1.Q1. `0.015`

16.1.Q2. `31`

16.1.Q3. `15745024`

16.1.Q4. A — Scaling and MLOps make model work operational: count parameters, estimate cost, serve, monitor and roll back.

16.1.Q5. A — prediction comes from setup, rule and evidence, not from guessing.

16.1.Q6. Weight memory is only the start; training also needs activations, gradients and optimizer state.

16.1.Q7. Double the hidden width and recount the parameters before changing code.

Appendix A – Glossary

activation. A nonlinear function such as ReLU that lets networks model curved relationships.

array. A typed block of values with a shape.

autograd. PyTorch's automatic differentiation system.

backpropagation. Efficient gradient computation through a chain of operations.

broadcasting. NumPy's rule for stretching compatible shapes without copying by hand.

channel. A colour or feature dimension such as red, green and blue.

context window. The token budget available for prompt plus response.

convergence. The process of updates approaching a stable good solution.

convolution. A sliding-window operation that detects local patterns.

cosine similarity. A direction-based similarity score between vectors.

cross-validation. Repeated train/validation splits used to estimate generalisation.

data leakage. Letting validation/test information influence training preparation.

dataframe. A labelled table with rows and columns.

divergence. When updates grow unstable and loss gets worse.

dot product. A sum of pairwise products, central to similarity and neural layers.

drift. A change in real-world data that makes old model behavior less reliable.

dtype. The numeric representation, such as int64 or float32.

embedding. A vector representation of text, image or other data.

encoding. Turning categories or text into numeric representations.

f1. The harmonic mean of precision and recall.

feature. A model input column or derived signal.

feature map. The output grid produced by a filter or layer.

fit. The training step that learns parameters from data.

flops. Floating-point operations, a rough measure of compute cost.

gradient. The direction a parameter should move to reduce loss.

grounding. Tying a model response to supplied evidence instead of unsupported guessing.

groupby. Split rows by key, apply an operation, combine results.

index. Labels pandas uses to align rows and columns.

indexing. Selecting values by position, slice or boolean mask.

key. The vector used to score whether a token is relevant.

layer. A transformation from input activations to output activations.

learning rate. The step size used when updating parameters.

logit. The raw linear score before a sigmoid turns it into a probability.

loss. A number measuring how wrong a prediction is.

loss landscape. The surface showing loss for different parameter values.

matrix multiplication. The @ operation that composes linear transformations.

module. A reusable neural-network component with parameters.

n-gram. A sequence of n adjacent tokens.

nearest neighbour. The stored vector closest to a query vector.

parameter count. The number of learned weights and biases in a model.

pipeline. A repeatable chain of preprocessing and model steps.

pixel. One location in an image grid.

precision. Of predicted positives, the fraction that were truly positive.

predict. The inference step that produces outputs for new examples.

prompt. The instruction and context sent to a language model.

query. The vector asking what information is needed.

rag. Retrieval-augmented generation: fetch relevant context before answering.

recall. Of actual positives, the fraction the model found.

scaling. Transforming numeric ranges so one feature does not dominate by units alone.

series. One labelled column or vector in pandas.

serving. Running a model for real requests under latency and reliability constraints.

shape. The dimensions of an array, such as (rows, columns).

sigmoid. A function mapping any real value into the range 0..1.

softmax. A function turning scores into a probability-like distribution.

tensor. A multidimensional array used by PyTorch.

text feature. A numeric representation derived from text.

token. A piece of text treated as one unit, such as a word or subword.

training loop. The repeated sequence forward pass, loss, backward pass, optimizer step.

true positive. A positive case correctly predicted positive.

value. The information vector mixed according to attention weights.

vector index. A data structure built for fast similarity search.

vectorisation. Applying one operation across many values at once.

vocabulary. The set of known tokens.

weight matrix. The learned linear transformation inside a layer.

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