

Day 23 Plan and scaffold

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Choose a capstone project and describe it in one paragraph, with its features split into must have and nice to have.
- Sketch the project's data, and list its functions as type-hinted stubs with docstrings.
- Set up a project folder with a virtual environment and separate module files.
- Make the single simplest feature work end to end.

What to remember from earlier

- Type hints such as `def total(prices: list[float]) → float` say what a function takes and what it gives back. ([Day 19: Type hints and code style, 19.1 Annotating Parameters and Return Values](#))
- A docstring's first line says what a function does, written as a command such as "Return the total." ([Day 19: Type hints and code style, 19.8 Docstrings and Comments That Help](#))
- A function that ends without a return statement gives back `None`. ([Day 9: Functions, part 1, 9.5 Returning Versus Printing](#))
- Any `.py` file is a module, so a file beside it can import its functions. ([Day 13: Modules, packages and environments, 13.4 Splitting Your Code Across Files](#))
- `python -m venv .venv` makes a virtual environment, and activating it puts `(.venv)` at the start of the prompt. ([Day 13: Modules, packages and environments, 13.9 Virtual Environments](#))
- A data class (`@dataclass`) writes `__init__`, `__repr__`, and `__eq__` for a class that mostly holds data. ([Day 17: Classes and objects, part 2, 17.8 Data Classes with @dataclass](#))
- Saving data to a CSV or JSON file, and loading it at the start, keeps it between runs. ([Day 14: Files and data formats, 14.10 Keeping Data Between Runs](#))

Welcome to the last phase. For the next three days you build **one project of your own**, from an empty folder to a tested program with a README. Today you plan it and get its first feature working; on Day 24 you build the rest; on Day 25 you polish it and look back over the course.

There is **no new reading** in Phase 5. Everything you need was taught on Days 1 to 22, and each milestone below names the day to reread when you get stuck. There is also **no reference solution**: your project is your own, so nobody else's code would match it. The self-check list at the end of each day is how you judge your work.

A **capstone** is a final project that pulls together everything a course taught. It is bigger than Day 12's checkpoint project in two ways. You choose and plan it yourself, and it uses the tools of a real project: several modules, a virtual environment, files, error handling, and tests.

Today's hour, roughly:

Milestone	Minutes
23.1 Choose a project	5
23.2 Describe it	10

23.3 Sketch the data	5
23.4 List the functions	10
23.5 Create the folder	5
23.6 The simplest feature	25

If the plan runs long, cut it short rather than the last milestone. A plan is worth most once something runs.

23.1 Choosing a Project

Pick **one** of these four projects. Each brief says what the finished program does and which days taught the tools it needs.

Expense tracker. Record each thing you spend money on: the date, the amount, a category such as food or transport, and a short note. The expenses are saved to a CSV or JSON file, so they are still there next time. The program shows every expense and the total spent in each category and in each month. Uses classes or a data class (Days 16 and 17), CSV or JSON files (Day 14), dates (Day 21), and counting or grouping (Day 21).

Weather command-line tool. Run `python weather.py Lisbon` and see the weather there now and for the next few days, laid out neatly. Options change what it shows, such as how many days. It explains the problem, instead of crashing, when the city does not exist or the network is down. Uses a web API (Day 22), `argparse` (Day 21), f-string formatting (Day 3), and error handling (Day 15). Day 22's Putting it together built a small version; your capstone is your own program, with options and at least one feature that version lacks.

Quiz game. Ask questions read from a JSON question bank, check each answer, and keep score. At the end, compare the score with the high scores saved in a file, and save the new list. Uses classes (Days 16 and 17), JSON files (Day 14), `random` to shuffle (Day 13), and error handling (Day 15).

Log or text file analyzer. Read a log file or any text file named on the command line, pull out the parts that matter with a regular expression, count them, and write a report: the most common pages, words, or errors, and when they happened. Uses files (Day 14), `re` and `Counter` (Day 21), `argparse` (Day 21), and dates (Day 21). Day 21's Putting it together summarized a web server's log; your analyzer can read the same kind of file or a different one.

Choose the one you would actually use or show someone. Whichever you pick, you will have three hours with it: today, Day 24, and Day 25. All four are sized for that.

Make a folder for the project, named after it with underscores, such as `expense_tracker`. Unlike Days 1 to 22, the project keeps **one** folder for all three days.

23.2 Describing the Project

Before code, write down what you are building. Start a file called `notes.txt` in the project folder and write two things in it.

1. A one-paragraph description. Three to five sentences, in plain English, saying what the program does and who it is for. If you cannot say it in a paragraph, the project is not clear yet, and code will not make it clearer. This is your **project brief**, the same kind of short "what, not how" description that Day 12 started from, except that this time you write it.

2. A list of features, split in two. A **feature** is one thing the program lets its user do, such as "add an expense" or "show totals by month".

- A **must have** feature is one the program cannot do its job without. Remove it, and the brief's paragraph is no longer true.
- A **nice to have** feature would make the program better, but it can do its job without it. These wait until every must have works, and possibly forever.

Here is a description and feature list for the expense tracker. Yours will differ even if you chose the same project:

Expense tracker

A command-line program that keeps track of what I spend. I type in each expense with its date, amount, category, and a short note, and the program saves it to a file so it is still there next time. It lists every expense and shows the total spent in each category and in each month, so I can see where my money goes.

Must have

1. Add an expense: date, amount, category, note
2. Save expenses to a file, and load them when the program starts
3. List every expense
4. Show the total for each category
5. Show the total for each month

Nice to have

- Delete an expense
- A monthly budget for each category, with a warning when it is passed
- Show only one month's expenses

Day 24 gives you about 40 minutes for features, so aim for **three to five must haves**, each about one function and one menu choice or option. Number them in the order you will build them: features that later ones depend on come first (you cannot total expenses you cannot add).

KEY IDEA

A must have feature is one the program cannot do its job without; a nice to have waits until every must have works. Keep the must haves to a list you can finish in two hours.

WATCH OUT

Too many must haves. Ten must haves feel ambitious today and turn into ten half-working features on Day 25. A program that does four things reliably is better than one that does ten things sometimes. If your list is longer than five, move the ones you would miss least to nice to have.

PAUSE AND TRY**Question 1** Multiple choice

Sam's quiz game description says: "The game asks questions from a question bank file, tells me after each one whether I was right, and keeps my score." Which feature is a **must have** for that description?

- A. A time limit of 20 seconds for each question
- B. Loading the questions from the JSON question bank
- C. Colored text for right and wrong answers
- D. Choosing a difficulty level before the game starts

23.3 Sketching the Data

Every one of the projects is, underneath, some data and some things done to it. Decide on the data before the functions, because the functions' parameters and return values are made of it.

Answer three questions in `notes.txt`:

- What things does the program deal with?** Each kind of thing with a few named attributes is a candidate for a class, usually a data class (Day 17). Day 17's rule applies: a class is worth it when the same group of values travels together everywhere; a plain dictionary is enough for a quick, one-off grouping.
- Where do they live while the program runs?** Usually a list of those objects, created once (Day 12's rule: in `main()`, before any loop).
- What is saved to disk, and in what format?** CSV suits rows that all have the same columns; JSON suits nested data, such as a question with a list of choices (Day 14). Write down the file name and one example line or record.

The expense tracker's sketch:

Data

- Expense (a data class): `day (date)`, `amount (float, in dollars)`, `category (str)`, `note (str, may be empty)`
- While running: one list of Expense objects, made in `main()`
- On disk: `expenses.csv`, a header row, then one row per expense:
`day,amount,category,note`
`2026-10-15,12.50,food,lunch with Sam`

The date is saved as ISO text such as `2026-10-15`, because that is the layout `date.fromisoformat` reads back and it sorts correctly as text (Day 21).

Starting points for the other projects:

Project	Things in it	Saved to disk
Weather tool	One day's forecast (date, low, high, rain); the parsed options	Nothing required; maybe a saved response for tests (Day 22)
Quiz game	A question (prompt, choices, answer); a high score (name, score)	<code>questions.json</code> (only read) and <code>high_scores.json</code> (read, write)

Log analyzer	One log entry (time, kind, details), a namedtuple or a data class	The report file it writes; the log itself is only read
--------------	---	--

23.4 Listing the Functions

Now list the functions you expect to need, written as real Python but without their bodies. A **stub** is a function whose signature and docstring are written but whose body is not, yet. Each stub has type hints on every parameter and the return value (Day 19), and a docstring that says what it does, what it returns, and what it raises (Day 19). A function whose body is only its docstring is complete, legal Python: the docstring is a statement on a line by itself, so it does nothing when the function runs.

Split the functions the way Day 12 and Day 22 did:

- **Logic functions** take data in and return data out. They never call `input()` or `print()`. These are the ones Day 24's tests will check.
- **Storage functions** load and save the file.
- **One `main()`**, or a few small helpers it calls, talks to the user: it reads input or command-line options, calls the other functions, and prints.

The expense tracker's logic module, as stubs:

```

1  """Expense tracker: the data and the calculations."""
2
3  from dataclasses import dataclass
4  from datetime import date
5
6
7  @dataclass
8  class Expense:
9      """One amount of money spent."""
10
11     day: date
12     amount: float
13     category: str
14     note: str = ""
15
16
17 def parse_amount(text: str) -> float:
18     """Return the amount typed as text, in dollars.
19
20     Raises:
21         ValueError: If text is not a number, or is 0 or less.
22     """
23
24
25 def total_by_category(expenses: list[Expense]) -> dict[str, float]:
26     """Return the total spent in each category."""
27
28
29 def total_by_month(expenses: list[Expense]) -> dict[str, float]:
30     """Return the total spent in each month, keyed like "2026-10"."""

```

and its storage module:

```

1  """Expense tracker: saving and loading expenses.csv."""
2
3  from pathlib import Path
4
5  from expenses import Expense
6
7
8  def load_expenses(path: Path) -> list[Expense]:
9      """Return every expense saved at path, or [] if the file does not exist."""
10
11
12 def save_expenses(path: Path, expenses: list[Expense]) -> None:
13     """Write every expense to path as CSV, replacing what was there."""

```

Writing the stubs is planning that Python can read. The hints force you to decide what each function takes and gives back, and a mismatch between two plans, such as one function returning a dictionary that the next expects to be a list, shows up now instead of on Day 25. Copy your feature list next to the stubs and check that every must have is covered by at least one function.

A type checker (Day 19) reports each stub whose hint promises a value as missing a return, because its body returns nothing. That is expected today; the complaints disappear as you fill in the bodies, so they double as a to-do list.

KEY IDEA

Write each function's signature, type hints, and docstring before its body. The stubs are your plan in Python: they show what each function takes and returns, and which functions are logic, storage, or talking to the user.

PAUSE AND TRY

Question 2 Tracing

What does this program print?

stub_call.py

```

1  def total_spent(amounts: list[float]) -> float:
2      """Return the sum of the amounts, in dollars."""
3
4
5  print("Start")
6  print(total_spent([4.5, 12.0]))
7  print("End")

```

23.5 Creating the Project Folder

With the plan written, build the skeleton: the folder, its environment, and one file per module.

1. Make the virtual environment (Day 13), in a terminal opened in the project folder:

```
python -m venv .venv
source .venv/bin/activate
```

The second line is for macOS and Linux; on Windows use `.venv\Scripts\Activate.ps1` in PowerShell or `.venv\Scripts\activate.bat` in Command Prompt, as in 13.9. Check for `(.venv)` at the start of the prompt.

2. Install what the project needs to run. Only the weather tool needs a third-party package: `python -m pip install requests`, then `python -m pip freeze > requirements.txt` (Day 13). For the other three projects, the standard library is enough today; Day 24 installs `pytest`.

3. Create the module files and paste your stubs into them. The expense tracker's folder:

```
expense_tracker/
  .venv/           made by python -m venv; never edit it
  expenses.py      the Expense class and the calculations (logic)
  storage.py       load_expenses and save_expenses
  main.py          the menu: the only file that calls input()
  notes.txt        your description, feature list, and daily notes
```

Give `main.py` a `main()` function and finish it with `if __name__ == "__main__":` and a call to `main()` (Day 13), so that importing it from a test file later does not start the program.

4. Run it. `python main.py` should finish without an error, even though nothing is written yet. That one run proves the environment is active, every file is where `import` looks for it, and every stub is legal Python. An `ImportError` or `ModuleNotFoundError` now means a misspelled module name or a file in the wrong folder, and it is much easier to fix in a skeleton than in a finished program.

WATCH OUT

Naming a module after a standard library module. A file called `csv.py`, `json.py`, `random.py`, or `datetime.py` in your project folder is found **before** the standard library module of the same name, so `import csv` in `storage.py` imports your file instead (Day 13). The error then appears in a strange place, such as `AttributeError: module 'csv' has no attribute 'writer'`. Name modules after what they do in **your** project: `storage.py`, `questions.py`, `analyzer.py`.

23.6 The Simplest Feature End to End

The last milestone is the most important: make **one** feature work completely. A feature works **end to end** when it runs the whole way through the program: from where the data comes in (typing, a command-line option, a file, or the network), through your logic, to where it comes out (the screen or a file). Building one thin feature through every layer, before making any layer complete, finds problems in the plan while they are cheap to fix: an import that does not work, a file saved where the program does not look, or a function whose return value the next one cannot use.

Pick the **single simplest** must have that still touches every layer. Suggestions:

Project	Simplest feature end to end
Expense tracker	Add one expense, save it, and see it listed after restarting the program

Weather tool	Print the current temperature for the city named on the command line
Quiz game	Load the question bank and ask the first question, saying whether the answer is right
Log analyzer	Read the file named on the command line and print how many lines match your pattern

Fill in only the stubs that this feature needs, and keep each body as plain as you can. Bad input is Day 24's job; today, type only good input. For the expense tracker, that means:

- `main()` shows a two-choice menu (add, list) and quits on a third choice, as Day 12's menu did.
- Choosing add asks for the date, amount, category, and note, builds an `Expense` (with `date.fromisoformat` for the date and `float` for the amount), appends it to the list, and calls `save_expenses`.
- `save_expenses` writes a header row and one row per expense with `csv.writer` (Day 14).
- `load_expenses` reads the rows back with `csv.DictReader` and rebuilds each `Expense`; `main()` calls it once, at the start.
- Choosing list prints each expense on one line.

Test it the way the brief says it should work: run the program, add an expense, quit, run it again, and list. The expense must still be there. Open `expenses.csv` in your editor and check that it holds what you expect. If the second run lists nothing, check that `main()` calls `load_expenses` before the menu loop and that both functions use the same path.

When the feature works, stop. Write today's recap in `notes.txt`: what works, what surprised you, and which must have you will build first tomorrow.

KEY IDEA

Get one thin feature working through every layer, input to logic to output or file, before finishing any layer. A working skeleton shows where the plan was wrong while it is still cheap to change.

PAUSE AND TRY

Question 3 Short answer

Lee plans a quiz game and decides to write every function in `questions.py` completely, all eight of them, before writing `main.py` or loading the JSON file. Give two problems this order could hide until late, and what Lee should build first instead.

Go further

Day 23 has no new reading. If setting up the project sent you back to Day 13 or Day 17, these are good second explanations: The Python Tutorial: Modules, `venv`: Creation of virtual environments, and dataclasses: Data Classes.

Self-check

Check each statement. If the answer is "not yet," reread the linked milestone and the day it names, and fix that part of your project before Day 24.

- ☐ I chose one project and can say in one sentence what it does. Reread: 23.1 Choosing a Project
- ☐ I wrote a one-paragraph description and split my features into must have and nice to have, with no more than five must haves. Reread: 23.2 Describing the Project
- ☐ I wrote down what classes or dictionaries hold my data and what gets saved to disk, in which format. Reread: 23.3 Sketching the Data
- ☐ I listed my functions as stubs with type hints and docstrings, and every must have is covered by at least one. Reread: 23.4 Listing the Functions
- ☐ I can tell which of my functions are logic, which are storage, and which talk to the user. Reread: 23.4 Listing the Functions
- ☐ My project folder has a virtual environment and one file per module, and python main.py runs without an error. Reread: 23.5 Creating the Project Folder
- ☐ My simplest feature works end to end, and I wrote my recap for today in my notes file. Reread: 23.6 The Simplest Feature End to End

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. The description says the questions come from a question bank file. Without loading it, there are no questions and the description is false, so it is a must have.

- **A** is wrong: the description never mentions time. A time limit would be a nice addition, not part of the job.
- **C** is wrong: the game tells the player whether they were right with or without color.
- **D** is wrong: nothing in the description depends on difficulty levels.

Reread 23.2 *Describing the Project* if you chose differently.

Question 2

```
Start
None
End
```

- Lines 1 and 2 define `total_spent`. Its body is only the docstring, which is legal and does nothing when the function runs.
- Line 6 calls it. The body ends without a `return`, so the call gives back `None` (Day 9), and `print` shows `None`. The hint says `float`, but Python does not check hints when it runs a program (Day 19); only a type checker would complain.
- The program does not crash, so a file full of stubs can be imported and run while you fill the bodies in one at a time.

Reread 23.4 *Listing the Functions* if your output differed.

Question 3

Model answer: Lee could spend the whole hour on functions and still have no program that runs. Problems in the plan stay hidden until everything is joined together, for example that the JSON file's layout does not match what the functions expect (a list where they expect a dictionary), that `main.py` cannot import `questions.py` because of a name or folder mistake, or that a function returns something the next one cannot use. Lee should first build the simplest feature end to end: load the question bank, ask one question, and say whether the answer is right, then add the other functions one at a time.

A correct answer names at least two problems that only show up when the layers are joined (file format, imports, mismatched return values, or nothing runnable) and says to build one feature through every layer first.

Reread 23.6 *The Simplest Feature End to End* if your reasoning differed.