Week 4 — “Make It Reusable, Make It Safe”

Course: Scientific Programming with Python
Timebox: Tutorial 75 min · Exercise 5 ≤ 30 min · Exercises 6–10 = home tasks for ~1 week

Narrative setup

Your station’s scripts have grown. This week you’ll turn repeated steps into handy tools and protect them against messy inputs. By the end, you’ll have a tiny toolbox you can import anywhere on the project—and it won’t crash on the first odd reading.

Exercise 1 — Pocket Helper for Labels

Type: General. Estimated time: 6–8 min

Field notes need a compact line like `DAY:ID:NOTE`. Create a small, reusable way to produce that line from three values, in one place, so you can call it whenever you log.

What you must produce. One script that defines a reusable line maker and prints two sample lines.

Inputs/Outputs. Inputs: day ∈ `[1,31]`, id = short text (e.g., `CU-PHY-2`), note ≤ 30 chars (no colons). Output: `14:CU-PHY-2:install-started`.

Reflection. What repeated details did you avoid rewriting?
Why this matters. One source of truth keeps logs consistent.

Hint: define a parameterized function.

Exercise 2 — Named Options for Units

Type: Scientific. Estimated time: 8–10 min

Convert a temperature value (°C) into either K or °F depending on a small named option the caller provides, with a sensible default when none is given.

What you must produce. A callable converter you can invoke several times with or without the named option; print the results.

Inputs/Outputs. Input: `tC` ∈ `[-40, 60]`; option ∈ `{K, F}`; default: `K`. Output examples: `K=294.1` for `21.0`; `F=69.8` for `21.0`.

Reflection. How did a default reduce typing at call sites?
Why this matters. Named options make calls self-explanatory.

Hint: define a parameterized function with at least one default argument.

Exercise 3 — Flexible Line Joiner

Type: General. Estimated time: 8–10 min

Sometimes you need to stitch 2, 3, or 4 short tags into a single route string, with a chosen separator (default `/`). Build one flexible joiner that copes with a varying count of pieces.

What you must produce. A callable tool; demonstrate with 2–4 pieces and two separators.

Inputs/Outputs. Inputs: 2–4 words (letters/digits/`-`), optional separator defaulting to `/`. Output: a single line (e.g., `roof/west/sensor-A`) for an input: “roof”, “west”, “sensor-A”.

Reflection. What benefits come from accepting “any number” of items?
Why this matters. Flexible inputs reduce one-off helpers.

Hint: define a parameterized function that supports a default argument and a variable number of arguments.

Exercise 4 — Safe Distance Calculator

Type: Scientific. Estimated time: 10–12 min

Compute the distance between two points `(x1,y1)` and `(x2,y2)` from the lab map. If any value is not numeric text, print a gentle warning and return nothing rather than stopping the program.

What you must produce. A callable distance tool; show two demo calls: one valid e.g. "0", "0", "3", "4", one with a bad input e.g. "0", "oops", "3", "4" that triggers the warning but keeps the script(program) running.

Inputs/Outputs. Inputs: textual fields that parse to floats (range: `[-1e3, 1e3]`). Output: either `distance=...` with 2 dp, or `warning: invalid number`.

Reflection. Where should a warning appear so it helps but doesn’t derail the run?
Why this matters. Robust tools survive messy data.

Hint: define a parameterized function that uses exception handling.

Exercise 5 — Mid-Challenge

Type: Scientific. Estimated time: ≤30 min

Bundle three small tools (functions) into a mini toolbox file you can **import**: (1) the label maker from Ex-1; (2) the unit converter from Ex-2; (3) the safe distance from Ex-4. Then, in a separate Python script file, use that toolbox to produce: `LABEL | T_K=... | T_F=... | d(P1,P2)=...` for given inputs.

What you must produce. One small toolbox file + one demo script file calling it.

Inputs/Outputs. Inputs: day,id,note; `tC` in `[-40,60]`; two points in `[-1e3,1e3]`. Output: one summary line; warnings as in Ex-4 if needed.

Reflection. How did separating "toolbox" vs "use-site" improve clarity?
Why this matters. Reuse across notebooks prevents drift.

Hint: besides supporting the required functionality, create a module and them import in your Python script.

Exercise 6 — Who Sees What? (Home)

Type: General. Estimated time: 30–40 min

Create a short demo that shows how a value with the same name behaves inside an inner helper (function) vs. outside it. Print what each part "thinks" the name's value is, before and after a change in the inner part.

What you must produce. A script that prints three labelled lines demonstrating name lookup order.

Inputs/Outputs. No inputs; output clearly shows "outside" and "inside" views.

Reflection. How can a local choice accidentally shadow a wider one?
Why this matters. Predictable naming avoids subtle bugs.

Exercise 7 — Calm Exit on Bad Rows (Home)

Type: Scientifi. Estimated time: 40–60 min

Read lines like `x1,y1,x2,y2` from `pairs.txt`. For each line, try to compute the distance. If a row is malformed, note it and continue; at the end, print how many rows succeeded and how many failed.

What you must produce. A script that opens a text file, processes all rows, and prints two counters.

Inputs/Outputs. File: 10–30 lines; numbers in `[-1e3,1e3]`; may contain blank or bad rows. Output: `ok=N`, `bad=M`.

Reflection. Where should the "try" be so one bad row doesn't stop the rest?
Why this matters. Recovery beats restart in long runs.

Hint: you are dealing with a text file pairs.txt. Use the strip() and split() functions of string object.

Exercise 8 — Little Toolbox as a File (Home)

Type: General. Estimated time: 40–60 min

Move your joiner from Ex-3 into its own file. In a second file, import it in two different ways and show both call styles in action.

What you must produce. Two files: one toolbox file and one caller; print three joined routes.

Inputs/Outputs. Inputs: your chosen tokens; default separator applies when omitted.

Reflection. What changed at the call site when you imported differently?
Why this matters. Modular code keeps projects tidy.

Exercise 9 — Gentle Stopper (Home)

Type: General. Estimated time: 30–45 min

Write a tiny checker that looks at a short string command (e.g., `ARM`, `DISARM`, `TEST`). If the command is unknown, raise a clear problem; in the caller, catch it and print a friendly message, then continue.

What you must produce. One checker function + a small driver script that calls it on 5 commands and never crashes.

Inputs/Outputs. Inputs: 5 tokens (some unknown), e.g. "ARM", "XXX", "TEST", "DISARM", "BAD". Output: action lines for known tokens; a friendly line for unknown ones.

Reflection. Why raise in one place and handle in another?
Why this matters. Clear fault paths aid maintenance.

Hint: Use exception handling and raise ValueError for unknown tokens.

Exercise 10 — Last-Resort Cleanup (Home)

Type: General. Estimated time: 45–60 min

Open an input file name given by the user and print the first 3 non-blank lines. Ensure that even if there is a problem (missing file, bad encoding), your code always runs its final cleanup message at the end.

What you must produce. A script that reports lines or a readable problem note, then prints `cleanup done`.

Inputs/Outputs. Input: file path string; Output: up to 3 lines (as-is) + `cleanup done`.

Reflection. Why is a guaranteed final step helpful for logs and files?
Why this matters. Predictable cleanup prevents resource leaks.