Lab 1 — Understanding Inherited Code

AI-Assisted Programming for PhD Researchers

You have inherited a field-study analysis from a PhD student who just graduated and left the group. There is one script, `legacy_analysis.py`, and a folder of raw measurements. Your supervisor wants the summary statistics by the end of the week. The catch: the script does not run, you did not write it, and you do not yet know what it actually computes.

In this lab you will get the script running and, more importantly, understand it well enough to trust (or distrust) its output. You will use the agent to explain the code, but you will verify its claims yourself. Reading someone else's code is a skill the agent cannot do for you.

Budget about **35 minutes**. Work in the starter repository you cloned during the Git lecture (Lecture II).

Task 1 — Run It (and Watch It Fail)

Goal: run the script, read the error, and understand why it fails.

From the repository root, run the script. It imports `matplotlib`, so run it with `uv` so the dependency is available:

```
uv run --with matplotlib python legacy_analysis.py
```

The import succeeds, and then the script stops with a traceback ending in a `FileNotFoundError`. Read the whole traceback, from bottom to top. The last line names the exception; the lines above it show where in the code it happened.

The offending line opens the data file:

```
f = open("/Users/postdoc-old-laptop/projects/field_study/measurements.csv")
```

That path existed on the previous student's laptop. It does not exist on yours.

> Checkpoint
>
> Running the script produces a `FileNotFoundError` pointing at an absolute path under `/Users/postdoc-old-laptop/...`. Nothing has been changed yet. You have only run the script and read the error.

⚠ Explain it

1. What exact file path does the script expect to find, and where does that data file actually live in your clone?
2. Why is a hardcoded absolute path a problem for anyone who is not the original author, even if the file were present?
3. The script got *past* the `matplotlib` import before it failed. What does that tell you about the order in which Python runs the file?

Task 2 — Make the Agent Explain It

Goal: get a fast, structured explanation of the whole script, then check it against the code yourself.

Start the agent in the repository root and switch it to **plan mode** (press `Tab` to toggle). Plan mode lets the agent read and reason without editing files.

```
opencode
```

Paste this prompt:

ℹ Prompt

Read `legacy_analysis.py` and `data/raw/measurements.csv` (first 30 lines). Explain: (1) what the script computes, (2) what assumptions it makes about the data, (3) every bug or fragile spot you can find, ranked by severity.

Read the agent's answer slowly. Do not accept it as correct yet. Treat it as a set of claims you now have to verify.

Checkpoint

The agent's ranked list should include at least: the temperature loop mixes Fahrenheit and Celsius readings (it ignores the `temp_unit` column), the humidity mean counts the `-99` missing-value sentinel as a real reading, the file paths are hardcoded absolute paths, and the temperature loop is duplicated (once for the mean, once for the plot). If the agent missed any of these, ask it to look again at the specific line.

⚠ Explain it

1. Which of the agent's findings did you confirm *yourself* by reading the code, and which did you take on trust? Name at least one you verified line-by-line.
2. The Fahrenheit/Celsius bug: on which line does it happen, and what single fact about the data does that line ignore?
3. The script prints an average temperature of roughly 22. **Work out for yourself** what the true average is, in °C: about one in four readings is stored in °F and left uncorrected, so a value near 12°C is counted as if it were near 54. Reason about which direction that pushes the mean and by roughly how much, then state your estimate of the real average before moving on. (It is closer to 12°C than to 22; convince yourself why.)

Task 3 — Standing Instructions

Goal: give the agent a short, project-specific rulebook so every future session starts from the same ground truth.

Still in the agent, run the built-in initializer:

```
/init
```

This scans the repository and writes an `AGENTS.md` file, the standing instructions the agent reads at the start of every session. Open the generated file and read it critically. `/init` tends to over-describe: it lists obvious things, guesses at intent, and pads with detail no one needs.

Edit `AGENTS.md` down to three things:

- **Project purpose**: one line.
- **Commands**: how to run and (later) test the code, using `uv`.
- **Rules**: add these two explicitly:
  - Treat `data/raw/` as read-only. Never modify or overwrite files there.
  - Explain your changes before committing.

Then commit the file:

```
git add AGENTS.md
git commit -m "Add project instructions for the agent"
```

Checkpoint

`AGENTS.md` is committed, is under about 10 lines, and contains a one-line purpose, the `uv` commands, and the read-only / explain-before-commit rules.

⚠ Explain it

1. Name one thing `/init` got wrong, overstated, or invented about this project.
2. Why is "treat `data/raw/` as read-only" worth writing down for an agent, when you would never think to write it down for a human collaborator?

Task 4 — First Fix

Goal: make the script run from the repository root on any machine, and nothing more. Leave the statistics bugs alone for now.

Switch the agent to **plan mode** again and paste:

ℹ Prompt

Plan the minimal change so the script runs from the repo root on any machine: read `data/raw/measurements.csv` relative to the repo, write the plot to `out/plot.png`, and create `out/` if it is missing. Don't fix the statistics bugs yet.

Read the plan before you let it touch anything. It should propose using a repo-relative path (for example, derived from `__file__`), changing the `savefig` target to `out/plot.png`, and creating `out/` if needed. If the plan also tries to fix the temperature or humidity bugs, tell it to stop. That is the next lab, and mixing unrelated changes into one commit makes them hard to review.

When the plan looks right, switch to **build mode** (`Tab`) and let the agent apply it. Then run the script:

```
uv run --with matplotlib python legacy_analysis.py
```

It now runs to completion. It prints an average temperature and an average humidity (both still wrong, as you established in Task 2) and writes `out/plot.png`.

Review the change hunk by hunk before committing. `git add -p` shows you each change and asks whether to stage it, so you actually see what the agent did:

```
git add -p
git commit -m "Read data and write plot with repo-relative paths"
```

Make sure `out/` is ignored so the generated plot never gets committed. Check that `.gitignore` lists `out/` (add it if the agent did not).

Checkpoint

The script runs from the repository root, reads `data/raw/measurements.csv` relative to the repo, and writes `out/plot.png` (with `out/` gitignored). The statistics are unchanged and still wrong. Your repository now matches the reference state for this lab.

If you fell behind, catch up with:

```
git fetch --tags && git checkout tags/lab-01-done -b catchup-01
```

Explain it

1. Why did we deliberately *not* fix the temperature and humidity bugs in the same change?
2. What did reviewing the diff with `git add -p` show you that simply trusting "the agent said it fixed the paths" would not have?

If You Are Ahead

Try the **explain-then-regenerate** move: a check on whether you actually understand the fix or only recognize it once you see it [1].

1. In your own words, write **one paragraph** describing exactly what the Task 4 fix should do: where the data comes from, where the plot goes, and what has to happen if `out/` does not exist. Do not look at your diff while writing it.
2. Start a **fresh** agent session (close the current one). Give it *only* your paragraph (not the original prompt from Task 4, not your diff) and let it implement the change on a clean copy of the script.
3. Diff the agent's result against your Task 4 commit. Where do they differ?

If the two agree, your explanation was complete enough to reproduce the work. If the fresh agent produced something wrong or incomplete, the gap is in your paragraph, not in the agent: the explanation was missing something you thought you understood. Rewrite the paragraph until it would produce the right fix.

Note

That's it for this lab. If anything is unclear, ask now — the next session builds on this state. If you fell behind, use the checkpoint tag from the last checkpoint box to catch up.

Bibliography

[1] D. H. Smith and C. Zilles, “Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions,” in *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, Milan Italy: ACM, July 2024, pp. 171–177. doi: 10.1145/3649217.3653582.