The Reproducible Student Researcher

The Reproducible Student Researcher

Data, Code, and Records That Survive Your Project

Gaurav Tiwari

gauravtiwari.org

First edition, 2026.

gauravtiwari.org

To everyone who has opened a folder called `final`
and found four files in it.

Contents

Preface

You will lose a week of work. The only question is when, and how much of it you get back.

It usually happens like this. Eight months into a project, you need to regenerate Figure 3 with one more data point. You open the folder. There are four scripts with promising names, two of them called something like `analysis_final` and `analysis_final_v2`, and you cannot remember which one made the figure. You run both. Neither produces the numbers in your draft.

Nothing was deleted. Nothing was corrupted. You simply have no record of what you did, and that turns out to be the same thing as not having done it.

Who this is for

You write code for your project, and nobody taught you how.

Maybe it's Python for a physics simulation, R for a psychology experiment, MATLAB for a signals project, or a very large spreadsheet. You got the science degree, not the software one. You've heard people mention version control and containers and vaguely feel you should be using them, and every tutorial you've found is written for someone building a web application.

This book is the practical subset, written for a student with a project and a deadline.

What it assumes, and what it doesn't

It assumes you can already write code that works. It says nothing about how to write better algorithms.

It does not assume you use Git, know what a commit is, work on Linux, or have any interest in becoming a software engineer. Where a tool is mentioned, its purpose is explained before its commands.

Examples are mostly shell and Python, with R alongside where the two differ, because those are what most student projects use now. Appendix E covers MATLAB and Julia. The habits transfer to any language, and to projects with no code at all.

Two ideas the whole book rests on

Reproducibility is a property of your records, not your intelligence. Losing work is not carelessness. It's the default outcome of a process with no record-keeping, and it happens to careful people constantly. The fix is mechanical.

Your first collaborator is yourself in six months. Every rule here sounds like it's for sharing work with strangers, and the person it actually protects is you, on the day the revision arrives and you have to rerun something you last touched a year ago. That person has forgotten everything and cannot be emailed.

How to use it

If you're at the start of a project, read Parts I and II now and set up the structure in Appendix A. An afternoon, and it's the cheapest insurance in this book.

If you're in the middle of one and it's a mess, start with Chapter 6 and Chapter 13. Those two fix the most damage per hour, and neither requires you to restructure anything.

If you're near the end and somebody has asked for your data and code, go to Part VI.

Gaurav Tiwari
gauravtiwari.org

Part I

Why Work Disappears

1

The Week You Lost

Nobody plans to lose work. It goes missing through ordinary, sensible actions taken in a hurry.

This chapter is the catalogue of how, because you can't defend against a failure you haven't named.

1.1 The six ways a project falls apart

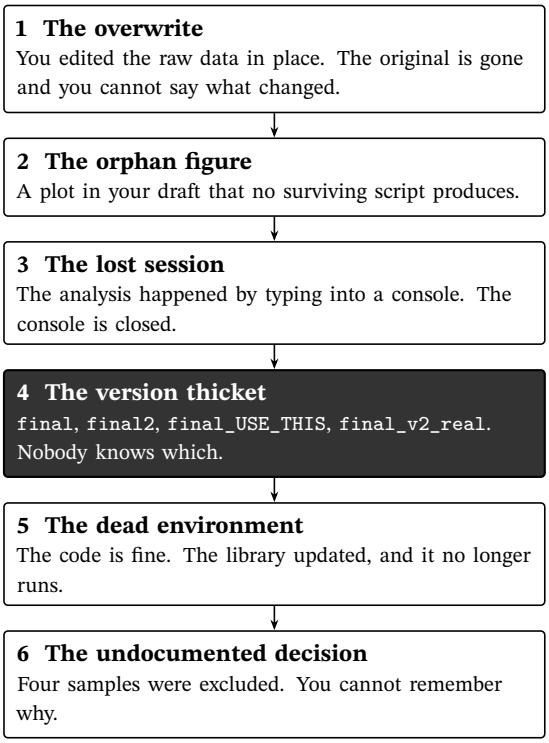

Figure 1.1: The six common failures. Note that only one of them involves anything being deleted. The other five leave every file intact.

Look at that list again. Only failure 1 destroys anything. In the other five, every byte you ever wrote is still on the disk, and the work is lost anyway, because what's missing is the record of what you did.

> *Losing work almost never means losing files. It means losing the link between a result and the process that produced it. Backups do not fix this, which is why people with good backups still lose weeks.*

1.2 Failure 1: the overwrite

The single most damaging one, and the easiest to prevent.

FRAGILE

Open `measurements.csv` in Excel. Notice three rows have the temperature in Fahrenheit. Fix them. Save. Continue.

REPRODUCIBLE

Leave `data/raw/measurements.csv` untouched, permanently. Write a script that reads it, converts those three rows, and writes `data/processed/measurements_clean.csv`. Commit the script.

The second is three minutes more work and it gives you something the first can never have: an answer to the question "what exactly did you change?". Which is the question a supervisor, a referee, or your own future self will eventually ask.

Chapter 6 is entirely about this rule.

1.3 Failure 3: the lost session

You open a Python or R console, load the data, try a few things, find something that works, and paste the number into your draft.

The number is now unreproducible. Not hard to reproduce, *impossible*, because the sequence of commands existed only in a console buffer that has since scrolled away or been closed.

This is the commonest way a student's analysis becomes irreproducible, and it's invisible while it's happening because the work feels productive. Chapter 13 covers the fix, which is not "stop using the console" but something narrower and easier.

1.4 Failure 4: the version thicket

```
analysis.py
analysis_v2.py
analysis_v2_fixed.py
analysis_final.py
analysis_final_ACTUAL.py
analysis_final_ACTUAL_2.py
```

Everyone recognises this and most people think it's a discipline problem. It isn't. It's what happens when you need to keep old versions and have no mechanism for it, so you invent one out of filenames.

The mechanism already exists, it's version control, and Part III covers the small fraction of it a student needs. The whole thicket collapses to one file called `analysis.py` with its history stored elsewhere.

1.5 Failure 5: the dead environment

Eighteen months later you run the code and get a stack trace. Nothing you wrote has changed. A library was updated, a function was renamed or its default argument changed, and your analysis now errors out or, worse, silently produces different numbers.

Silently is the real danger. A crash tells you something is wrong; a changed default gives you a plausible number that doesn't match your thesis.

Chapter 15 is about recording the environment, which takes one command and is the step almost everyone skips.

1.6 Failure 6: the undocumented decision

Every project accumulates decisions. Four samples excluded. One outlier removed. A threshold set at 0.3. A detector channel ignored because it was noisy.

Each was sensible at the time and took ten seconds. Eight months later they're indistinguishable from arbitrary choices, and if you can't say why you excluded something, a referee is entitled to assume you excluded it because it was inconvenient.

Chapter 7 covers the lab-notebook habit that fixes this, which costs about ninety seconds a day.

WHAT ACTUALLY BREAKS

The failure I've watched cost students the most is not any single one of these. It's the combination of 2 and 6 arriving together, eight months apart from the work: a figure nothing reproduces, showing data filtered by a rule nobody wrote down. At that point the honest options are to redo the analysis from raw data, which takes weeks, or to publish something you cannot fully defend. Neither is a good afternoon.

1.7 Why this isn't a character flaw

Students who lose work are not lazy. They're usually the ones moving fastest, because speed is exactly what generates these failures: the console session instead of the script, the quick edit in the spreadsheet, the copy of the file "just for now".

Every one of those is locally rational. Each saves five minutes today. The cost lands months later, on a different day, and by then the connection to the decision is invisible.

> *Every shortcut in this chapter saves minutes now and costs days later, and the delay between the two is what makes them so hard to stop taking. The defence is a habit, not a resolution.*

1.8 What the rest of the book does

Six habits, each cheap, each preventing one or more of the failures above.

Habit	Prevents
Raw data is read-only	1
Everything is a script	2, 3
Version control	4
Record the environment	5
Keep a lab notebook	6
One command rebuilds everything	2, 3, and most of the rest

None of them takes more than an afternoon to set up. Together they are the difference between a project you can defend and one you can only describe.

DO THIS TODAY

Open your current project folder right now and look for evidence of each of the six failures. Most people find three. Write down which ones you have; those chapters are the ones to read first, and the rest of the book will still be there when you need it.

2

What Reproducible Actually Means

The word gets used for four different things, and mixing them up is why the advice sounds impossible.

Only one of the four is your responsibility as a student, and it's the easiest.

2.1 The four levels

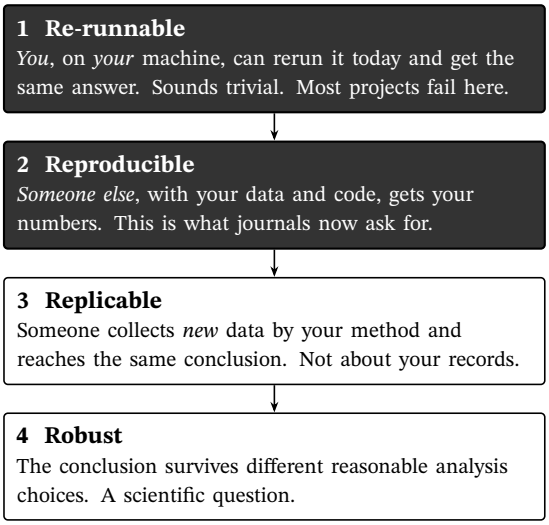

Figure 2.1: The four levels, from cheapest to hardest. A student project owes levels 1 and 2. Level 4 is a scientific question, not a records question.

Levels 3 and 4 are about the science and are mostly outside your control. Levels 1 and 2 are about your records and are entirely within it.

This book is about levels 1 and 2. That's not a modest goal: most student projects, and a distressing share of published papers, fail level 1.

> *If you cannot rerun your own analysis on your own machine today and get the same numbers, nothing else in this book matters yet. Start there.*

2.2 The test for level 1

Concrete, and you can run it this afternoon.

1. Pick a number or a figure from your current draft.
2. Without looking at your notes, regenerate it from raw data.
3. Time yourself.

Under 5 minutes	You have level 1. Move to Part VI and package it
Under an hour	Close. The pipeline exists but isn't automated; Chapter 14
Over an hour	The steps exist in your head, not on disk. Parts II and IV
Can't do it	The common case, and the reason for this book

Most students who try this the first time land in the third or fourth row, and are surprised, because the work itself was done carefully. Careful work and reproducible work are different properties.

2.3 The test for level 2

Harder, and worth doing once before you submit anything.

Copy your project to a fresh folder, or better, a different machine, or best, ask a labmate. Follow only your own written instructions. Regenerate one figure.

Whatever breaks is what would have broken for a referee, an examiner, or a future employer. Common results:

A missing file	It was somewhere else on your machine, not in the project
An absolute path	`/Users/you/Desktop/data.csv` exists on exactly one computer
A missing library	Installed once, years ago, never recorded
A manual step	"Then I opened it in Excel and sorted by column C"
A different number	An unseeded random process, or a library version change

All five are cheap to fix once you know they're there. All five are invisible until you test.

2.4 What reproducible does not mean

Three misconceptions that make people give up before starting.

It doesn't mean anyone can understand your code. Readable code is good and it's a different goal. Ugly code that runs and produces the stated numbers is reproducible.

It doesn't mean your result is correct. A reproducible analysis can be reproducibly wrong. What reproducibility buys is that the error is *findable*, which is not nothing: an irreproducible error is permanent.

It doesn't require containers, continuous integration, or a software-engineering course. Those exist and most student projects need none of them. A folder, a script, a requirements file, and a README will get you to level 2.

WHAT ACTUALLY BREAKS

The gap between what's advertised and what's needed is why students bounce off this topic. Search for reproducible research and you'll find advice about Docker images, workflow engines, and continuous integration pipelines. That advice is for a lab maintaining a tool used by hundreds of people. For a one-person project with a deadline,

the entire return is in about six habits, and the fancy tooling adds almost nothing on top.

2.5 What you actually owe, by situation

Coursework	Level 1. You need to rerun it when the marker asks a question
Final-year project	Level 1, plus a README, because it gets handed over
Master's thesis	Level 2. An examiner may ask, and the next student inherits it
A paper	Level 2, with data and code deposited. Increasingly mandatory
Anything with a funder	Check the mandate. Most now require deposited data

2.6 The cost, honestly

Setting this up costs about a day, spread across a project.

Habit	Setup	Ongoing
Project structure	30 min	none
Raw data read-only	10 min	none
Version control	1 hour	2 min a day
Environment file	5 min	1 min a month
Lab notebook	none	90 sec a day
Build script	2 hours	occasional
README	1 hour	10 min a month

Against that, one lost week is forty hours. The arithmetic isn't close, and it stays not close even if you only ever lose a week once.

DO THIS TODAY

Run the level 1 test today, properly, with a timer: pick one number from your draft and regenerate it from raw data. Write down how long it took and what stopped you. That list is your reading order for the rest of this book.

3

The Five Artefacts

A reproducible project is five things kept in a known state.

Everything in the rest of this book is about one of the five, so it's worth having the map before the detail.

3.1 The five

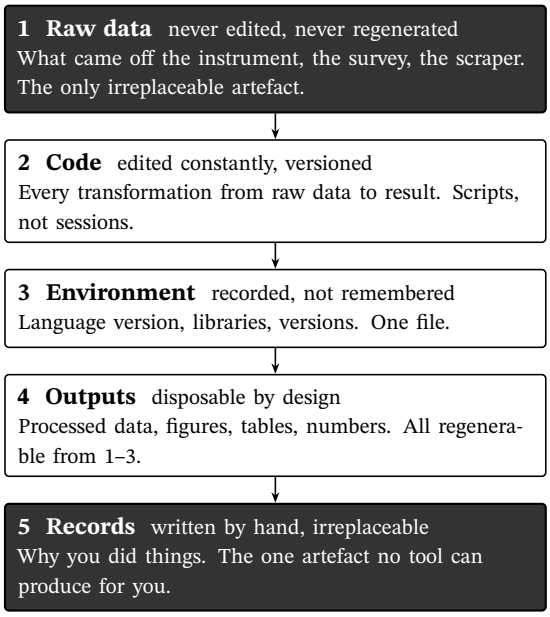

Figure 3.1: The five artefacts and the direction they flow. Raw data never changes; everything to its right is regenerable. The records sit alongside, describing choices the code cannot.

Two of the five are irreplaceable: the raw data and the records. The other three can be rebuilt, and the code can be recovered from version control even if you delete it.

That asymmetry decides everything about how you treat them.

> *Protect what cannot be regenerated. Raw data and written records get backups, redundancy, and paranoia. Outputs get deleted freely, because deleting them is how you find out whether you can rebuild them.*

3.2 Artefact 1: raw data

Whatever arrived from the world. Instrument files, survey exports, scraped pages, photographs, transcripts, a spreadsheet a collaborator emailed you.

Three rules, and Chapter 6 elaborates all three:

- **Never edit it.** Not once, not to fix an obvious typo.
- **Never generate it from code.** If code made it, it's an output.
- **Keep the provenance with it.** Where it came from, when, and under what settings.

That last point is the one people skip. A folder of `.csv` files with no record of which instrument, which day, or which calibration is raw data you cannot use in a year.

3.3 Artefact 2: code

Every transformation between raw data and a result.

The word to watch is *every*. If one step happened by hand, in a spreadsheet or a GUI or a console, the chain is broken there, and a broken chain means the output cannot be regenerated.

Students underestimate how many manual steps there are. A typical project has three or four, each of which felt too small to script.

3.4 Artefact 3: the environment

The language version, the libraries, and their versions.

This is the artefact people forget exists, because it's invisible while it works. Your code doesn't mention that it needs pandas 2.1; it just imports pandas, and one day pandas is version 4 and the behaviour has changed.

Recording it takes one command. Chapter 15.

3.5 Artefact 4: outputs

Processed data, figures, tables, fitted parameters, the numbers in your draft.

The counterintuitive rule: **outputs are disposable**. You should be able to delete the entire outputs folder and rebuild it with one command. If that thought frightens you, you have found something valuable, because the fear is precise information about which step isn't automated.

WHAT ACTUALLY BREAKS

Deleting your outputs folder and rebuilding it is the single best test in this book, and almost nobody does it voluntarily. Do it on a copy the first time if you must. Whatever fails to rebuild is the exact gap between what you think your pipeline does and what it does, and that gap is invisible by any other means.

3.6 Artefact 5: records

Why you did things.

No tool produces this. Version control records that you changed a threshold from 0.5 to 0.3; nothing but you can record that you changed it because the detector's noise floor turned out to be higher than the datasheet claimed.

That's the sentence a referee will ask for, and it's the one that evaporates within about three weeks if it isn't written down. Chapter 7.

3.7 How they map onto folders

The whole structure follows from the five artefacts, which is why the project template in Appendix A looks the way it does.

```
project/
  data/
    raw/          artefact 1. read-only, backed up, never edited
    processed/    artefact 4. regenerable, safe to delete
  code/           artefact 2. version controlled
  results/
    figures/      artefact 4. regenerable
    tables/       artefact 4. regenerable
  notes/          artefact 5. the lab notebook, decisions, dead ends
  environment.yml artefact 3. one file
  README.md       how to run all of it
```

You can rename any of these. What matters is that the five artefacts are separated, and specifically that **raw and processed data are in different folders**, because that separation is what makes the read-only rule enforceable.

3.8 The one-line summary

Artefact	Changes?	If you lose it
Raw data	Never	The project is over
Code	Constantly	Recoverable from version control
Environment	Rarely	Painful, and reconstructable
Outputs	Every run	Rebuild in one command
Records	Grows	Irreplaceable, and nobody notices until asked

DO THIS TODAY

Sort your current project folder into the five artefacts, on paper, not by moving files yet. Anything you can't classify is worth a second look: it's usually an output somebody has been treating as raw data, which is the most dangerous confusion of the five.

Part II

Files and Records

4

Naming Things

The cheapest habit in the book, and the one that pays back first.

A filename is metadata. It's the only metadata that survives being emailed, copied to a USB stick, downloaded twice, and found in a folder three years later.

4.1 Three rules

Machine-readable	No spaces. No punctuation beyond hyphen, underscore, and dot. Consistent field order
Human-readable	Someone who doesn't know the project can guess what it is
Sorts correctly	Dates as `YYYY-MM-DD`. Numbers zero-padded

FRAGILE

```
Final data (copy).xlsx
run 3 - 2nd attempt!.csv
plot for meeting.png
Untitled1.py
data 12/3/26.csv
```

REPRODUCIBLE

```
2026-03-12_run03_temperature-sweep_raw.csv
2026-03-12_run03_temperature-sweep_clean.csv
fig02_retention-vs-cycles.png
fit_two-mechanism-model.py
```

Everything in the second block sorts chronologically, groups by run, says what it contains, and can be typed into a terminal without quoting.

That last point matters more than it sounds. A space in a filename breaks naive shell scripts, and a slash in a date is illegal on every operating system, which is why the first block's last line silently became something else when it was saved.

4.2 The date rule

Use `YYYY-MM-DD`. Always. It's the ISO 8601 standard and it has one property nothing else has: alphabetical order is chronological order.

`2026-03-12`	Sorts correctly. Unambiguous worldwide
`12-03-2026`	Sorts by day. Is it 12 March or 3 December?
`12/03/26`	Illegal in a filename, and ambiguous
`March12`	Sorts April before March

The ambiguity row is not pedantry. A British student and an American collaborator will read `03-12` differently, and the error is undetectable from the file.

4.3 Field order, and why it matters

Put the fields in the order you'd want to group by, most general first.

```
2026-03-12_run03_temperature-sweep_raw.csv
|          |     |                 |
date       run   what              stage
```

Sorted alphabetically, that groups by date, then run, then measurement. If you'd rather group all temperature sweeps together regardless of date, put the measurement first. There's no universal right answer, and there is a right answer for your project. Pick it once.

4.4 Zero-pad your numbers

Without padding, files from `run1` through `run10` sort as 1, 10, 2, 3. With two digits, `run01` through `run10` sort correctly up to 99. Use three digits if you might exceed that.

This costs one character and prevents an entire category of confusion, including scripts that process files in the wrong order and produce plausible, wrong results.

4.5 Versions do not go in filenames

```
analysis_v2_final_ACTUAL.py
```

This is version control implemented badly in filenames. The real thing does it properly, and Part III covers it.

The exception, and it's a real one: **data files that genuinely are different versions of a dataset**. A survey re-export with corrected entries is a new file, and it should say so: `2026-04-02_survey-responses_v2.csv`, with a note in the lab notebook saying what changed. That's provenance, not version control.

4.6 Naming conventions for the common types

Raw data	`DATE_source_what_raw.ext`
Processed	Same stem, `_clean` or `_processed`
Scripts	Verb first: `clean_survey.py`, `fit_model.py`, `make_fig02.py`
Figures	`figNN_what-it-shows.png`, numbered as in the manuscript
Tables	`tabNN_what-it-shows.csv`
Notes	`YYYY-MM-DD.md`, one per working day

The verb-first rule for scripts is worth adopting. A folder of `clean_`, `fit_`, `plot_`, `make_` tells you the pipeline at a glance, whereas `survey.py` and `model.py` tell you nothing about what runs when.

4.7 Figures named for the manuscript

Name each figure file for the number it has in the paper, and keep the script that makes it named to match.

```
code/make_fig02.py   ->   results/figures/fig02_retention-vs-cycles.png
```

When a referee asks for a change to Figure 2, you know exactly which script to open. Without this, finding the right script is a ten-minute archaeology exercise, every time.

When the figures get renumbered during revision, rename both together. It's a two-minute job that stays cheap only if you do it immediately.

WHAT ACTUALLY BREAKS

The names that cause the most trouble are the ones that were meaningful for about a fortnight. `new_data.csv` is unambiguous on the day you create it and useless three months later, when there are two things that could be described as new. The test is not "does this make sense to me now?" but "will this be the only file this description fits, in a year?"

4.8 Things that break filenames

Spaces	Break shell commands unless quoted. Use hyphens
`/`	Illegal. Silently mangled by some software
`: * ? " <` `> |`	Illegal on Windows; your collaborator has Windows
Accents, emoji	Encoding trouble when moving between systems
Case	macOS treats `Data.csv` and `data.csv` as the same file; Linux does not. Use lowercase
Very long names	Some tools truncate past 255 characters, including the path
Leading dots	Hidden files on Unix, which is occasionally what you want and usually a surprise

That case row bites people specifically when a project moves from a Mac to a cluster and two files that coexisted on Linux collapse into one.

DO THIS TODAY

Write your naming convention down, in the README, in three lines: the field order, the date format, and the padding. Then rename one folder of files to match. Do the rest as you touch them; a bulk rename of a live project is how references get broken.

5

A Project Layout That Survives

One folder structure, set up in half an hour, used for every project you ever do.

The value isn't in any particular layout. It's in having the same one every time, so that finding things stops being a decision.

5.1 The layout

```
project-name/
  README.md              what this is, how to run it
  environment.yml        or requirements.txt / renv.lock
  .gitignore             what version control ignores
  Makefile               or run_all.sh
  data/
    raw/                 READ-ONLY. never edited, never generated
    processed/           generated. safe to delete
    external/            reference data from elsewhere, with provenance
  code/
    01_clean.py
    02_fit.py
    03_make_figures.py
    utils.py
  results/
    figures/             generated
    tables/              generated
    logs/                generated
  notes/
    2026-03-12.md        the lab notebook, one file per working day
    decisions.md         the running log of choices and why
  docs/
    protocol.md          how the data was collected
```

That's the whole thing. Appendix A has it as a script you can run to create it.

5.2 Why these particular divisions

Each boundary encodes a rule.

`raw` vs `processed`	Makes the read-only rule enforceable. You can set permissions on one folder
`data` vs `results`	Inputs versus conclusions. Different backup and sharing rules
`code` separate	So version control can track it without tracking gigabytes of data
`notes` separate	Records are an artefact, not an afterthought
Everything generated in two places	So "delete and rebuild" is a two-line command

> *Every folder is either wholly regenerable or wholly irreplaceable. Never mix the two, because a folder containing both cannot be safely deleted and cannot be safely ignored.*

5.3 Numbered scripts

The `01_`, `02_`, `03_` prefix does one job: it tells a stranger, and you in a year, what order things run in.

```
01_clean.py        raw/ -> processed/
02_fit.py          processed/ -> results/tables/
03_make_figures.py processed/ + tables/ -> results/figures/
```

Three scripts, an obvious order, and the data flows one way. When you outgrow this, Chapter 14 covers build tools, and the numbering is still what you'd write in the README.

Leave gaps if you like: `01`, `05`, `10`. Inserting a step later then doesn't renumber everything.

5.4 Paths: relative, always

The single most common reason a project doesn't run on another machine.

FRAGILE

```
project = "/Users/gaurav/Desktop/project"
path = project + "/data/raw/survey.csv"
df = pd.read_csv(path)
```

REPRODUCIBLE

```
from pathlib import Path
ROOT = Path(__file__).resolve().parents[1]
df = pd.read_csv(ROOT / "data" / "raw" / "survey.csv")
```

The second works on any machine, from any working directory, for any user. The `parents[1]` walks up from `code/` to the project root, so the script doesn't care where it's launched from.

In R, the `here` package does the same job:

```
library(here)
df <- read.csv(here("data", "raw", "survey.csv"))
```

Both idioms take one line and eliminate an entire category of failure.

5.5 One project, one folder

Everything for the project lives inside the project folder. No data on the Desktop, no scripts in Downloads, no figures in a shared drive somewhere else.

The test: could you zip this one folder and hand over the whole project? If the answer needs a "yes, but also…", the exception is the thing that will be missing when it matters.

The legitimate exceptions are large raw datasets that genuinely can't live in the folder. Handle those by putting a small text file in `data/raw/` that says where the data is, how big it is, and how to fetch it. Chapter 6.

5.6 Naming the project folder

`2026_msc-thesis_perovskite-cycling`	Year, kind, subject. Sorts and describes
`Project`	You will have four of these
`Untitled folder 3`	Yes, really

5.7 What goes at the top level

Only files that describe or drive the whole project: the README, the environment file, the build script, the licence, and version control's config. Everything else goes in a subfolder.

A cluttered top level is the first sign a project is drifting, and it happens through exactly one mechanism: somebody saves a file without choosing where.

WHAT ACTUALLY BREAKS

The layout matters less than its constancy. A researcher who uses the same imperfect structure for every project finds things instantly, because knowing where to look has become automatic. Someone who designs a beautiful bespoke structure per project spends thirty seconds orienting every single time they come back, and after two years has six structures and remembers none of them.

5.8 Adapting it

The layout above suits an analysis project. Adjust for what you do.

Simulation work	`data/raw/` becomes `config/`: the parameter files are your inputs
Wet lab	Add `docs/protocols/` and keep instrument exports in `data/raw/`
Fieldwork	Add `data/raw/photos/` and a `data/raw/manifest.csv` recording what was collected where
Theory	`code/` may be a `calculations/` folder of notebooks or algebra scripts; the rest is unchanged
Software	Follow your language's conventions instead; this layout is for analysis, not for a package

DO THIS TODAY

Create the structure for your current project today, empty, alongside what you have. Then move files into it as you touch them over the next fortnight rather than all at once, because a big-bang reorganisation breaks every path in every script on the same afternoon.

6

Raw Data Is Read-Only

If you take one rule from this book, take this one.

Raw data is never edited. Not to fix a typo, not to remove an outlier, not to convert units, not once, not ever.

6.1 Why it's absolute

Because the alternative destroys the only thing you cannot regenerate.

Code can be rewritten. Figures can be remade. The environment can be rebuilt. Raw data came from an instrument, a survey, or a moment in time, and if you overwrite it, that state is gone permanently.

There's a second reason, and it's the one that actually costs students marks: an edited raw file has no record of what changed. You know you fixed three temperatures. You do not know, six months later, which three, or what they were before, or whether you also fixed a fourth.

> *Every change to data happens in code, so that the change is recorded, reversible, and inspectable. A change made by hand is a change nobody can audit, including you.*

6.2 What to do instead

The pattern is always the same: read raw, transform in code, write processed.

FRAGILE

Open `measurements.csv`. Three rows are in Fahrenheit. Fix them by hand. Save. Delete the outlier in row 47 while you're there.

REPRODUCIBLE

```
raw = pd.read_csv(ROOT / "data/raw/measurements.csv")

# Rows 12, 18, 31 were logged in Fahrenheit: the datalogger was
# reset on 2026-03-04 and lost its unit setting. See
    ↪ notes/2026-03-04.md
mask = raw["run_id"].isin([12, 18, 31])
raw.loc[mask, "temp_c"] = (raw.loc[mask, "temp_c"] - 32) * 5 / 9

# Run 47 excluded: thermocouple detached mid-run, confirmed in the
# instrument log. Exclusion rule fixed before analysis began.
clean = raw[raw["run_id"] != 47]
```

```
clean.to_csv(ROOT / "data/processed/measurements_clean.csv",
    ↪ index=False)
```

Look at what the second version gives you that the first cannot. The original file is intact. The change is explicit and reversible. The *reason* is recorded next to the change. And when a referee asks how many runs were excluded and why, the answer is a line of code with a comment, not a memory.

6.3 Making it physically read-only

Willpower is not a mechanism. Set the permissions.

```
# macOS and Linux: make everything in raw/ read-only
chmod -R a-w data/raw/

# Windows PowerShell
Get-ChildItem -Recurse data\raw | ForEach-Object { $_.IsReadOnly = $true }
```

Now an accidental save fails with an error instead of succeeding silently. That error is the whole point: it converts an invisible mistake into a visible one.

To add new raw data later, unlock, add, relock. The friction is deliberate and it's about ten seconds.

6.4 Provenance travels with the data

A folder of CSV files with no context is not usable raw data. Put a plain-text file beside them.

```
data/raw/README.md
```

```
# Raw data

## measurements_2026-03-12.csv
Source:      Keithley 2450 SourceMeter, lab B14
Collected:   2026-03-12, 09:20–17:40, by G. Tiwari
Settings:    4-wire sense, NPLC 1, 10 mV compliance
Calibration: 2026-01-08 (cert. in docs/calibration/)
Columns:     run_id, t_s, temp_c, voltage_v, current_a
Notes:       Datalogger reset at 14:10; runs 12, 18, 31 logged
             in Fahrenheit. Corrected in code/01_clean.py, not here.
Rows:        4,812
Checksum:    sha256 3f9a...  (see checksums.txt)
```

Ten minutes when the data arrives. Impossible to reconstruct in a year.

6.5 Checksums, cheaply

A checksum tells you whether a file has changed. It costs one command and detects both accidental edits and silent corruption during copying.

```
# Record, once, when the data arrives
shasum -a 256 data/raw/* > data/raw/checksums.txt

# Verify, any time later
shasum -a 256 -c data/raw/checksums.txt
```

Run the verification before any analysis you're going to publish. It takes seconds and it's the only way to detect that a file changed without anybody meaning to.

6.6 When raw data is too big for the project folder

Common with imaging, sequencing, simulation output, and video.

Leave a stub in `data/raw/` recording everything needed to fetch it:

```
# data/raw/WHERE_IS_THE_DATA.md

The 340 GB image stack is not in this repository.

Location:  university research store, /rds/projects/perovskite/2026-03/
Backup:    LTO tape, box 14, requested via research-computing@
Fetch:     code/00_fetch_data.sh  (needs VPN + your university login)
Size:      340 GB, 12,400 files
Checksums: data/raw/checksums.txt covers all of them
```

The fetch script matters. "It's on the shared drive somewhere" is not provenance, and the person who knows where will have graduated.

6.7 Spreadsheets, specifically

Spreadsheets are where raw data goes to be silently corrupted, so they get their own warnings.

Autocorrected genes	Excel converts gene names like SEPT2 and MARCH1 to dates. A genuine, documented, widespread problem in biology
Leading zeros	Postcodes, IDs and phone numbers lose them silently
Date reformatting	Ambiguous dates get reinterpreted on open, per locale
Long numbers	Beyond 15 digits, precision is dropped without warning
Invisible formulas	A cell showing 4.2 may be computed from cells you later change
Row limits	Old formats truncate at 65,536 rows. Silently

If your data arrives as a spreadsheet, export it to CSV immediately, *check the export*, and treat the CSV as the raw file. Then never open the CSV in a spreadsheet program again, because opening and saving is enough to trigger several of the above.

WHAT ACTUALLY BREAKS

The gene-name problem is worth knowing about even outside biology, because of what it demonstrates. It persisted for years, in published supplementary files, in a field full of careful scientists. Not one of them decided to corrupt their data. They opened a file, and the software helpfully changed it, and nothing told them. That is what a silent tool does to raw data, and it is exactly why the rule is absolute rather than a preference.

6.8 The exceptions, and how to handle them

Two situations look like editing raw data and aren't.

Genuinely corrupt files. If a file is truncated or unreadable, keep it, and record the problem. Do not delete it; a corrupt file is evidence about what happened.

Anonymisation. Where ethics requires identifiers to be stripped before storage, the anonymised file becomes your raw data. Record who did the stripping, when, and by what procedure, and keep the identifiable original wherever the ethics approval says it must live, which is usually not your laptop.

DO THIS TODAY

Set `data/raw/` to read-only today, write the provenance README for whatever is already in there, and generate the checksum file. Half an hour, and it closes the failure that ends projects.

7

The Digital Lab Notebook

Ninety seconds a day, and it's the artefact no tool can generate for you.

Version control records *what* changed. The notebook records *why*, and why is what everybody asks for later.

7.1 What goes in it

Not everything. Five things.

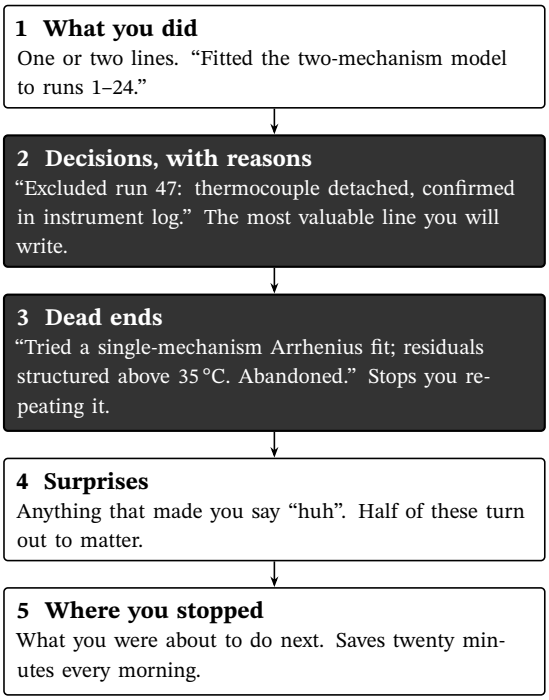

Figure 7.1: The five things worth writing down. Notice that four of the five are about decisions and dead ends, which is exactly what no automated tool captures.

Items 2 and 3 are the ones with real value. Item 2 is what a referee asks for. Item 3 is what stops you spending a second week on something you already ruled out in March, which is a thing that genuinely happens.

> *Code records what you did. The notebook records what you decided and what failed. Neither can be reconstructed from the other, and the second evaporates within about three weeks.*

7.2 The format

Plain text, one file per working day, in the project folder.

```
notes/2026-03-12.md
```

```
# 2026-03-12

## Did
- Ran temperature sweep, runs 12-31, 25 to 45 C in 5 C steps.
- Cleaned into data/processed/measurements_clean.csv (01_clean.py).

## Decided
- Excluded run 47. Thermocouple detached at ~t=140 s; the instrument
  log shows the open-circuit flag. Exclusion rule was fixed before
  I looked at the fitted values.
- Fitting temp in Celsius, not Kelvin. Arbitrary, but the Arrhenius
  code expects K, so 02_fit.py converts. Flagging in case of sign
  confusion later.

## Dead end
- Single-mechanism Arrhenius fit. Looks fine to 35 C, residuals go
  structured above it. Not a fitting problem, tried three initial
  guesses. Suggests a second mechanism, which is the whole result.

## Surprised by
- Retention at 45 C is 9 points below Kumar 2023. They used
  cylindrical cells at 0.5C; ours are pouch at 1C. Could be either.

## Next
- Fit the two-mechanism model, runs 1-24 only, hold out 25-31.
```

Four minutes to write. Consider how much of that is recoverable from the code alone: the first section, and nothing else.

7.3 Why plain text, in the project folder

Plain text	Opens in fifty years. No app, no account, no export
In the project	Travels with the code. Gets backed up with it
Version controlled	Free history, and you can see what you knew when
Searchable	`grep` finds every mention of run 47 in a second
One file per day	No merge conflicts, no giant unnavigable file

Notion, OneNote, and Evernote are fine tools and they're the wrong place for this. They live outside the project, they need an account, and exporting five years of notes from a proprietary format is a job nobody does until the subscription lapses.

7.4 The running decisions file

Alongside the daily notes, keep one cumulative file.

```
notes/decisions.md
```

```
# Decisions

| Date       | Decision                          | Why |
|------------|-----------------------------------|-----|
| 2026-02-08 | Exclusion: initial capacity <95% of nameplate | Below spec;
    ↪ rule fixed before cycling began |
| 2026-03-12 | Excluded run 47                   | Thermocouple detached,
    ↪ instrument log confirms |
| 2026-03-19 | Two-mechanism model, not one      | Structured residuals above
    ↪ 35 C |
| 2026-04-02 | Fit on runs 1-24, hold out 25-31 | Avoid fitting and testing
    ↪ on the same data |
```

This is the file you open when writing the methods section, and it turns a two-day archaeology job into an hour.

The date column matters more than it looks. "The exclusion rule was fixed before cycling began" is a claim, and a dated entry from before the results existed is evidence for it.

7.5 When to write it

At the end of each working session, not the end of the week.

The half-life of "why did I do that" is about three days. By Friday you can reconstruct Monday's decisions only approximately, and approximately is the same as not at all for the purpose they serve.

Set a reminder if you need one. Most people find that after a fortnight the habit sticks on its own, because they've already had the experience of looking something up and finding it there.

WHAT ACTUALLY BREAKS

The entry students value most, months later, is almost always a dead end. Not the successful analysis, which is in the code anyway, but the paragraph saying "I tried this and it didn't work, and here's how I know". It stops the repeat attempt, and it often turns out to be a paragraph in the discussion section, because a documented failure is evidence about the system.

7.6 Notebooks for wet labs and fieldwork

If your institution requires a physical bound notebook, keep it. This is in addition, not instead.

The split that works: the physical notebook records what happened at the bench, with dates and signatures if required. The digital one records what happened to the data afterwards, which is the part the bench notebook can't hold.

Photograph the physical pages into `docs/notebook-scans/` at the end of each week. Cheap insurance, and it makes the bench record searchable alongside everything else.

7.7 The minimum viable version

If five sections a day sounds like too much, do this instead:

```
2026-03-12: excluded run 47, thermocouple detached (instrument log).
            single-mechanism fit fails above 35 C, residuals structured.
```

Two lines, thirty seconds, in one running file. That's about eighty percent of the value, and a habit you actually keep beats a template you abandon in April.

DO THIS TODAY

Write today's entry before you close your laptop, even if the day was unproductive, and especially if something failed. Then write one retrospective entry for the most important decision you have already made on this project, while you can still remember why.

8

Backups, and the 3-2-1 Rule

Version control is not a backup. Cloud sync is not a backup. This chapter is short because the rule is short.

8.1 The rule

Three copies, on two kinds of media, one of them somewhere else.

3	Copies of anything irreplaceable. The working copy counts as one
2	Different media or services, so one failure mode can't take both
1	Off-site, so a fire, a theft, or a flood can't take everything

For a student project that's usually: the laptop, an external drive, and either the university's research store or a cloud account. Fifteen minutes to set up.

8.2 Why sync isn't backup

Dropbox, OneDrive, Google Drive and iCloud propagate your mistakes. That's their job.

Delete a folder, and it's deleted everywhere within seconds. Corrupt a file, and the corrupt version syncs. Encrypt everything with ransomware, and the encrypted files sync.

Most of these services keep version history, typically 30 days, and it will save you if you notice quickly. It will not save you from discovering in November that something went wrong in June.

> *A backup is a copy that does not change when the original changes. If deleting the original removes the copy too, that's replication, not backup.*

8.3 Why version control isn't backup either

Git protects your code, which is the artefact you're least likely to lose in bulk.

It doesn't hold your raw data, because data doesn't belong in a repository. If your repository is on GitHub and your laptop dies, you have your code and none of your measurements.

8.4 What actually needs backing up

The two irreplaceable artefacts from Chapter 3, and nothing else urgently.

Artefact	Priority	Why
Raw data	Critical	Cannot be regenerated at any price
Notes	Critical	Cannot be regenerated, and nobody notices until asked
Code	Important	In version control, plus its remote
Environment file	Important	Tiny, in version control
Processed data	Low	Rebuild from raw
Figures, tables	Low	Rebuild from code

This is why the folder layout matters: if raw data and notes are in two known places, backing up the right things is a two-line command instead of a judgement call.

8.5 A backup that takes one command

```
# Mirror the irreplaceable parts to an external drive.
# --archive keeps timestamps and permissions; --delete is deliberately
# NOT used, so a deletion at source does not propagate.
rsync -av data/raw/ /Volumes/Backup/project/data/raw/
rsync -av notes/    /Volumes/Backup/project/notes/
```

Note the missing `-delete`. That flag makes the backup a mirror, which means an accidental deletion propagates on the next run. Leaving it off means the backup accumulates, which is what you want.

Put those two lines in `backup.sh` and run it weekly. Add the date to the destination if you want generations:

```
DEST=/Volumes/Backup/project/$(date +%Y-%m-%d)
rsync -av --link-dest=/Volumes/Backup/project/latest data/raw/
    ↪ "$DEST/data/raw/"
ln -sfn "$DEST" /Volumes/Backup/project/latest
```

The `-link-dest` trick hard-links unchanged files, so each dated snapshot costs only the difference. You get a browsable history of every backup at almost no disk cost.

8.6 Use the university's storage

Most institutions provide research storage that is backed up properly, kept in a real data centre, and free to you.

Students routinely don't use it because nobody mentions it at induction. Ask your supervisor or the research computing team. It's usually the best off-site copy you'll get, and it often satisfies a funder's data-management requirement at the same time.

8.7 Test the restore

An untested backup is a hypothesis.

Once, early, restore one file from each backup to a scratch folder and check it opens. Ten minutes.

The failures this catches are dull and common: the external drive that was never actually mounted when the script ran, the cloud folder that stopped syncing in February, the archive that needs a password nobody recorded.

WHAT ACTUALLY BREAKS

The backup that fails is almost never absent. It's the one that stopped working silently and nobody noticed. A sync client signed out after an OS update, a drive that mounted read-only, a scheduled task that errors every week into a log nobody reads. Which is why the only meaningful test is restoring a file, not checking that a backup exists.

8.8 Before anything irreversible

Take a fresh backup before: reinstalling an operating system, a major software update, handing back a loaned laptop, travelling with the only copy, and any large-scale reorganisation of the project folder.

That last one catches people. A well-intentioned restructure at 11pm is a common way to lose a folder.

8.9 The five-minute setup

1. Buy or find an external drive bigger than your data.
2. Write `backup.sh` with the two `rsync` lines above.
3. Ask about university research storage; put a third copy there.
4. Run the script. Restore one file. Confirm it opens.
5. Set a weekly calendar reminder to run it.

Five steps, and it retires the entire category of disaster that ends projects outright.

DO THIS TODAY

Do step 4 today, on whatever backup you currently believe you have. Restore one raw data file to a scratch folder and open it. Most people discover something, and it is much better to discover it now.

Part III

Version Control

9

Git Without the Mythology

Git has a reputation for being hard. It is hard, if you learn it the way it's usually taught, which is as a distributed version control system with a content-addressable object database.

You need about five percent of it. This chapter is what a commit actually is, and the next one is the commands.

9.1 What problem it solves

The version thicket from Chapter 1:

```
analysis.py
analysis_v2.py
analysis_final.py
analysis_final_ACTUAL.py
```

You made those files because you needed to keep old versions and had no mechanism for it. Git is the mechanism. You keep one file called `analysis.py`, and every past state of it is retrievable.

9.2 What a commit is

A commit is a **snapshot of your whole project at one moment, with a message saying what changed and why.**

That's it. Not a diff, not a patch, not a file. A snapshot of everything being tracked, plus a note.

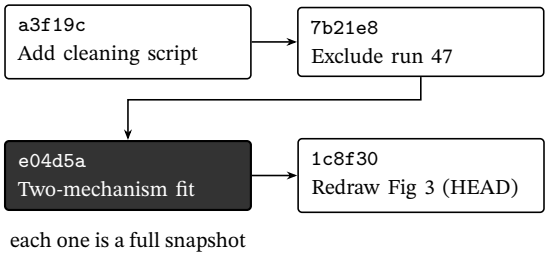

Figure 9.1: A repository is a chain of snapshots. Each has a message, an author, a timestamp, and a pointer to the one before it. You can return to any of them.

The six-character codes are commit hashes. You'll almost never type one, and it's useful to know they're how Git names snapshots.

> *A commit is a save point you can return to, with a note explaining why you made it. Everything else Git does is built on that one idea.*

9.3 The three places a file can be

The one piece of Git's model you genuinely have to hold in your head.

Working directory	The files on disk. What you edit
Staging area	Changes you've marked as going into the next commit
Repository	The committed history. Permanent

```
    edit files          git add            git commit
working directory  -->  staging area  -->  repository
```

The staging area is the part beginners find pointless, and it has one real use: committing some of your changes and not others. You fixed a bug and also left some debugging print statements around; stage the fix, commit it, deal with the rest separately.

If that sounds like more trouble than it's worth right now, use `git commit -a`, which stages and commits everything at once. Chapter 10 shows both.

9.4 What Git is good at, and what it isn't

Good at	**Bad at**
Text: code, notes, LaTeX, CSV	Large binary files
Showing exactly what changed	Word documents, PDFs, images
Returning to any past state	Anything over 100 MB
Recovering deleted files	Being a backup of your data

That right-hand column is why **data does not go in the repository**. Git stores a full copy of every version of every file, so a 2 GB dataset committed five times is 10 GB of history that everyone who clones the project has to download. Chapter 12 covers what to exclude.

9.5 The words

Repository, repo	The project folder plus its history
Commit	A snapshot, or the act of making one
Stage	Mark a change for the next commit
`HEAD`	Where you are now in the history
Branch	A named line of development. `main` by default
Remote	A copy elsewhere, usually on GitHub
Push	Send your commits to the remote
Pull	Fetch commits from the remote
Clone	Download a repository for the first time
Diff	The differences between two states

9.6 Do you need GitHub?

No, and you probably want it.

Git works entirely on your own machine. GitHub, GitLab and Bitbucket are websites that host a copy, which gives you an off-site copy of your code, a way to share it, and somewhere to point when a journal asks.

Private repository	Free on all the major hosts. Use one while the work is unpublished
Institutional GitLab	Many universities run one. Ask; it may be required for some data
Student accounts	GitHub Education gives students extras, free

A caution worth stating: pushing to a public repository publishes whatever is in it, permanently and immediately, and deleting it afterwards does not reliably remove it from forks and caches. Start private, and make it public deliberately when the work is ready.

WHAT ACTUALLY BREAKS

Almost every student who bounces off Git does so at the same place: they try to learn branching, merging, and rebasing before they have committed anything. Those are collaboration tools. For one person on one project, Git is a linear chain of save points, and that mode is genuinely simple. Learn the rest when a collaborator arrives, and not before.

9.7 What it costs

An hour to set up and understand. Then roughly two minutes a day: one command at the end of a working session.

Against that, it retires the version thicket permanently, gives you a searchable history of every change with reasons attached, lets you recover any deleted file, and hands you an off-site copy of your code for free.

DO THIS TODAY

Install Git and run `git config --global user.name` and `user.email` to identify yourself. That's the whole setup. The next chapter turns your current project into a repository in four commands.

10

The Seven Commands You Need

Git has about 150 commands. You will use seven.

10.1 Starting a repository

Once per project, in the project folder.

```
cd path/to/project
git init
```

That creates a hidden `.git` folder holding the history. Nothing else changes, and nothing is tracked yet.

Before your first commit, write a `.gitignore` (Chapter 12), because it is much easier to keep data out than to remove it later.

10.2 The seven

`git status`	What has changed. Run this constantly
`git add`	Stage changes for the next commit
`git commit`	Save a snapshot with a message
`git log`	See the history
`git diff`	See what changed, line by line
`git push`	Send commits to the remote
`git pull`	Fetch commits from the remote

`git status` is the one to overuse. It tells you what state you're in and usually suggests the command you want next. Running it before and after everything else is how people learn Git without reading a manual.

10.3 The daily loop

```
git status                        # what have I changed?
git add code/02_fit.py            # stage this file
git add notes/2026-03-12.md       # and this one
git commit -m "Fit two-mechanism model; exclude run 47"
git push                          # send it off-site
```

Or, when everything you changed belongs in one commit:

```
git commit -a -m "Fit two-mechanism model; exclude run 47"
git push
```

The `-a` stages every tracked file that changed. It does not stage brand-new files, which still need `git add` once.

That's the whole daily habit: two commands at the end of a session.

10.4 Writing a commit message

The message is the part with lasting value, and it's the part people waste.

FRAGILE

```
git commit -m "update"
git commit -m "fixes"
git commit -m "asdf"
git commit -m "final version"
```

REPRODUCIBLE

```
git commit -m "Exclude run 47: thermocouple detached mid-run"
git commit -m "Convert runs 12/18/31 from F to C; logger reset
    ↪ 2026-03-04"
git commit -m "Switch to two-mechanism fit; single-mechanism residuals
structured above 35 C"
```

Three rules. Say what changed and why. Write it in the imperative, as an instruction the commit carries out. Keep the first line under about seventy characters, and put detail on later lines if you need it.

> *Write the message for someone trying to find, six months from now, the commit where a specific decision was made. That someone is you, and `git log --grep="run 47"` only works if the messages say something.*

10.5 Reading the history

```
git log --oneline                # compact list
git log --oneline -10            # last ten
git log --grep="run 47"          # search messages
git log -p code/02_fit.py        # every change to one file, with diffs
git log --since="2026-03-01"     # by date
```

`git log -p` on a single file is the one that earns its keep. It shows you every change ever made to that script, newest first, with the message attached. That's the answer to "when did this threshold change, and why".

10.6 Seeing what changed

```
git diff                  # unstaged changes
git diff --staged         # what's about to be committed
git diff HEAD~1           # against the previous commit
```

Run `git diff --staged` before every commit. It takes five seconds and it catches the debugging print statement, the hard-coded path, and the commented-out block you meant to delete.

10.7 Undoing things

The reason people are frightened of Git, and the reason it's worth having.

You want to	Command
Discard changes to a file	`git restore FILE`
Unstage a file	`git restore --staged FILE`
Fix the last commit message	`git commit --amend`
Recover a deleted file	`git restore FILE`
Get an old version of a file	`git restore --source=HEAD~3 FILE`
Undo a commit, keep the changes	`git reset --soft HEAD~1`
See a file as it was	`git show HEAD~5:code/02_fit.py`

One genuine warning. `git restore FILE` discards your uncommitted changes to that file, permanently, with no confirmation. It is the one command in this chapter that can lose work. Everything committed is recoverable; anything never committed is not.

Do not use `git reset --hard` until you understand it. It is the command in every panicked forum answer and it deletes uncommitted work across the whole project.

10.8 Connecting a remote

Create an empty repository on GitHub, then:

```
git remote add origin https://github.com/you/project-name.git
git push -u origin main
```

The `-u` sets the default, so afterwards `git push` alone is enough.

10.9 How often to commit

When something works	The most useful trigger. A commit is a state you can return to
Before anything risky	A refactor, a big change, a library upgrade
End of every session	Even if unfinished. Say so in the message
Not every keystroke	A commit per line makes the history unreadable
Not once a month	A commit spanning three weeks records nothing useful

Two to ten commits a working day is a reasonable rhythm.

WHAT ACTUALLY BREAKS

The habit that separates people who benefit from Git from people who merely use it is committing when something *starts working*, not when they stop for the day. The first gives you a history of working states you can return to. The second gives you a history of Fridays.

10.10 The graphical option

If the terminal is a barrier, GitHub Desktop, GitKraken, Sourcetree, and the Git panels built into VS Code, RStudio, and PyCharm all do everything in this chapter with buttons.

Use them. The model in Chapter 9 is what matters, and the interface is a preference. RStudio's Git panel in particular is how most R users work and it's entirely sufficient.

DO THIS TODAY

Turn your current project into a repository today: write `.gitignore`, then `git init`, `git add .`, `git commit`. Then make one more commit at the end of every working session this week. After five days the habit is most of the way formed.

11

Branches, and Whether You Need One

Probably not. This chapter is short on purpose.

11.1 The honest answer

If you are one person working on one project, you can use Git productively for three years without ever creating a branch.

Branching is a collaboration tool. It solves the problem of several people changing the same files at once, and of keeping an unfinished feature away from a working version that other people depend on. A student project usually has neither problem.

> *Work on `main`. Commit often. Add branches when you have an actual reason, and the reason will be obvious when it arrives.*

Most Git tutorials open with branching because they're written for software teams. Skipping it is not cutting a corner; it's using the right subset.

11.2 What a branch is

A movable label pointing at a commit.

When you create a branch and commit on it, the history forks: your new commits are on the branch, and `main` still points where it did. You can move between them, and later merge one into the other.

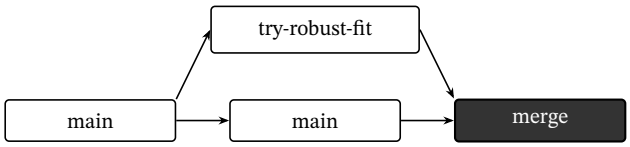

Figure 11.1: A branch is a second line of commits from a shared point. Merging brings the work back together.

11.3 The three times a student genuinely wants one

A risky experiment. You want to try restructuring the analysis and you're not sure it'll work.

```
git switch -c try-robust-fit    # create and move to a new branch
```

```
# ... work, commit as usual ...
git switch main                 # abandon it: main is untouched
git branch -D try-robust-fit    # delete it if it went nowhere
```

Honestly, committing on `main` handles this too, because you can always go back to an earlier commit. The branch is tidier if the experiment runs for a fortnight.

A collaborator. Two people editing the same scripts. Then branches, and merges, are what they're for, and it's worth an hour with a proper tutorial at that point.

A frozen version. You submitted the thesis and want to keep working. Tag it rather than branch it:

```
git tag -a thesis-submitted -m "Version submitted 2026-09-01"
git push --tags
```

A tag is a permanent label on one commit. That's exactly what you want for "the state of the code when I submitted", and it's the right thing to cite in a paper or archive.

11.4 If you do branch, the minimum

```
git switch -c branch-name       # create and switch
git switch main                 # switch back
git branch                      # list branches
git merge branch-name           # bring the work into main
git branch -d branch-name       # delete after merging
```

Five commands. `git switch` is the modern spelling; older material uses `git checkout` for the same job, and both still work.

11.5 Merge conflicts, briefly

When two lines of work change the same lines, Git asks you to decide. It marks the file:

```
<<<<<<< HEAD
threshold = 0.3
=======
threshold = 0.5
>>>>>>> try-robust-fit
```

Edit the file so it says what you want, delete all three marker lines, then `git add` it and `git commit`. That's the whole procedure, and it's much less alarming than its reputation.

If it goes badly, `git merge --abort` returns you to where you started.

WHAT ACTUALLY BREAKS

Conflicts feel like something has gone wrong, and they're the system working. Git is refusing to guess which of two changes you meant, which is exactly what you'd want it to do. The tools that silently pick one, and some do, are the ones that lose work quietly.

11.6 What to do instead of branching

For a solo project, these give most of the benefit with none of the complexity:

Commit when things work	Every working state is a return point
Tag milestones	`thesis-submitted`, `paper-v1`, `data-frozen`
Write real messages	So you can find the commit you want
`git log` before panicking	The state you want is usually already in the history

DO THIS TODAY

Tag your current state with something meaningful, even if it's just `before-cleanup`. Then carry on committing to `main` and come back to this chapter when a second person joins the project.

12

What Never Goes in a Repository

Three categories: data, secrets, and generated files. One of the three can end your project.

12.1 The `.gitignore` file

A plain-text file at the top of the project listing what Git should pretend doesn't exist. Write it before your first commit.

```
# ---- data ----------------------------------------------
data/raw/*
!data/raw/README.md
!data/raw/checksums.txt
data/processed/*
!data/processed/.gitkeep

# ---- generated outputs ---------------------------------
results/figures/*
results/tables/*
results/logs/*
!results/**/.gitkeep

# ---- secrets -------------------------------------------
.env
*.key
*.pem
credentials.json
secrets.yml

# ---- environment and caches ----------------------------
__pycache__/
*.pyc
.venv/
venv/
.Rhistory
.RData
.Rproj.user/
.ipynb_checkpoints/
node_modules/

# ---- operating system junk -----------------------------
.DS_Store
Thumbs.db
desktop.ini

# ---- editor --------------------------------------------
```

```
.vscode/
.idea/
*.swp
```

The `!` lines are exceptions: ignore everything in `data/raw/` *except* the README and the checksums, both of which are small, textual, and exactly what you want in the history.

12.2 Why data stays out

Git stores every version of every tracked file forever. Commit a 2 GB dataset five times and the repository is 10 GB, permanently, for everyone who ever clones it.

Worse, removing it later is genuinely hard. Deleting the file in a new commit doesn't shrink the history; the old versions are still in there. Purging them requires rewriting history, which breaks every existing clone.

> *It is easy to keep data out of a repository and hard to get it out later. Write `.gitignore` before `git init`, not after.*

The exception: small, textual, rarely-changing reference data. A 40 kB CSV of site coordinates that never changes is fine and useful in the repository. The test is size, format, and change rate, not principle.

12.3 Secrets, and why this one is serious

API keys, passwords, database credentials, access tokens, participant identifiers.

WHAT ACTUALLY BREAKS

Pushing a secret to a public repository should be treated as an immediate compromise, not a mistake to quietly fix. Automated scanners find newly committed credentials within minutes; this is a well-documented, continuously running activity. Deleting the file in a later commit does nothing, because the old commit is still in the history, and forks and caches keep copies you cannot reach. The only safe response is to revoke and reissue the credential.

The pattern that avoids it entirely: put secrets in a file the repository ignores, and read them at runtime.

```
# .env   -- listed in .gitignore, never committed
API_KEY=sk-live-abc123
DB_PASSWORD=hunter2
```

```
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.environ["API_KEY"]   # crashes loudly if it's missing
```

Then commit a `.env.example` with the keys and no values, so a collaborator knows what they need to supply.

For participant data specifically, this is an ethics matter as well as a technical one. Identifiable data in a repository is a breach even if the repository is private, because private repositories are shared, forked, and backed up in ways you don't control.

12.4 Generated files stay out

Figures, processed data, compiled output, logs. All regenerable from code and raw data, so committing them adds bulk and creates confusion about which version is authoritative.

They also produce constant meaningless changes: re-run the analysis and every PNG differs at the byte level even when the plot is identical, so `git status` fills with noise and you stop reading it.

The exception worth making: **the final figures for a submitted paper**. Commit those once, at submission, and tag the commit. They're a record of what you actually sent.

12.5 Keeping empty folders

Git tracks files, not folders, so an empty `results/figures/` vanishes on clone and your script fails trying to write into it.

The convention is a placeholder:

```
touch results/figures/.gitkeep
```

with a `!` exception in `.gitignore`, as above. Alternatively have your scripts create their output directories, which is more robust:

```
(ROOT / "results" / "figures").mkdir(parents=True, exist_ok=True)
```

12.6 Checking before you commit

```
git status                  # anything here you didn't expect?
git add .
git status                  # what is actually staged?
git diff --staged           # read it before committing
```

Ten seconds, and it's how you catch the 800 MB file and the credentials before they're permanent rather than after.

To check the damage in an existing repository:

```
# the ten largest files ever committed
git rev-list --objects --all \
  | git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize)
    ↪ %(rest)' \
  | awk '$1=="blob" {print $3, $4}' | sort -rn | head -10
```

12.7 If something is already in there

Committed, not pushed	`git rm --cached FILE`, add to `.gitignore`, amend or commit
Pushed, private repo, large file	Tolerable. Add to `.gitignore` and move on unless the size is genuinely a problem
Pushed, a secret	**Revoke the credential now.** Then clean the history. In that order
Pushed, participant data	Revoke access, tell your supervisor and your data-protection officer today

History rewriting, if you need it, is done with `git filter-repo` or the BFG Repo-Cleaner. Both are well documented, both break every existing clone, and neither is a substitute for revoking a leaked credential.

DO THIS TODAY

Write `.gitignore` before you run `git init`. If your repository already exists, run the large-file check above and search the history for anything that looks like a key: `git log -p | grep -i --"api_key|password|secret"`. Five minutes, and the failure it prevents is the expensive kind.

Part IV

Code That Runs Again

13

Scripts, Not Sessions

The most common way a student's analysis becomes irreproducible, and the one that feels most productive while it's happening.

13.1 The failure

You open a console. You load the data, try a transformation, plot it, try a different transformation, fit something, get a number that looks right, and paste it into your draft.

The number is now unreproducible. Not difficult, *impossible*, because the sequence that produced it existed only in a console buffer.

FRAGILE

```
>>> df = pd.read_csv("measurements.csv")
>>> df = df[df.temp > 20]
>>> df2 = df.groupby("run").mean()
>>> df2 = df2[df2.voltage < 4.2]      # why 4.2? nobody remembers
>>> np.polyfit(df2.cycles, df2.retention, 1)
array([-0.0412,  99.8])
```

That last array is in your thesis. Six months later you cannot say what the 4.2 was, whether the temperature filter came before or after the grouping, or whether you ran anything else in between that changed the state.

13.2 The fix, which is not what people expect

The fix is not "stop using the console". The console is the right tool for exploring, and exploring is most of research.

The fix is: **when something works, move it into a script, then run the script from scratch.**

REPRODUCIBLE

```
"""02_fit.py -- fit retention against cycle count.

Run:  python code/02_fit.py
In:   data/processed/measurements_clean.csv
Out:  results/tables/fit_params.csv
"""
from pathlib import Path
import pandas as pd, numpy as np
```

```
ROOT = Path(__file__).resolve().parents[1]

MIN_TEMP_C = 20.0
MAX_VOLTAGE_V = 4.2   # cells above this were still in the CC phase;
                      # see notes/2026-03-19.md

df = pd.read_csv(ROOT / "data/processed/measurements_clean.csv")
df = df[df.temp_c > MIN_TEMP_C]
per_run = df.groupby("run").mean(numeric_only=True)
per_run = per_run[per_run.voltage_v < MAX_VOLTAGE_V]

slope, intercept = np.polyfit(per_run.cycles, per_run.retention, 1)

out = pd.DataFrame({"slope": [slope], "intercept": [intercept],
                    "n_runs": [len(per_run)]})
out.to_csv(ROOT / "results/tables/fit_params.csv", index=False)
print(f"slope={slope:.4f}  intercept={intercept:.1f}
    ↪ n={len(per_run)}")
```

Same analysis. Now the order is fixed, the magic numbers are named constants with reasons attached, the inputs and outputs are declared at the top, and running it twice gives the same answer.

> *Explore in the console. Commit findings to a script. The script is the record; the console is scratch paper, and scratch paper gets thrown away.*

13.3 The rule that makes it work

Run your script from a clean state before you believe it.

```
python code/02_fit.py
```

Not by pasting it into an already-warm console. From the command line, in a fresh process, top to bottom.

This is the step people skip, and it's the one that catches the bug where your script depends on a variable you defined by hand twenty minutes ago and never wrote down. That bug is invisible from inside a session and fatal outside one.

In R, the equivalent is restarting the session and sourcing:

```
# Session > Restart R, then:
source("code/02_fit.R")
```

And turn off RStudio's "restore `.RData` on startup". It silently reloads your old workspace, which means your script appears to work because variables from last week are still in memory.

13.4 Anatomy of a script that will still run

Header comment	What it does, how to run it, what goes in, what comes out
Imports at the top	All of them, so a reader sees the dependencies
Paths from a root	Never absolute. Chapter 5
Constants named	`MAX_VOLTAGE_V = 4.2`, not `4.2` buried in a filter
One job per script	Clean, or fit, or plot. Not all three
Writes to a file	Not just prints. Printed output is lost
Reproducible order	No dependence on what you ran before

The named-constant rule earns more than it looks. A number that appears once, at the top, with a comment saying why, is a decision you can find and defend. The same number buried mid-file is a magic value nobody can audit, including you.

13.5 Print, or write?

Both, and they have different jobs.

Print	For you, while it runs. Progress, sanity checks
Write to a file	For the record. Anything that goes in the manuscript

Any number that ends up in your thesis should exist in a file on disk that a script produced. Chapter 19 is entirely about that, because retyping a printed number into a manuscript is where transcription errors live.

13.6 Functions, lightly

You do not need to become a software engineer. Two guidelines:

Repeat something three times and make it a function. Twice is fine. Three times means you'll eventually fix a bug in two of the three.

Put shared functions in `utils.py` and import them. Not copy-pasted between scripts, which is the same problem with extra steps.

```
# code/utils.py
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]

def load_clean():
    """The cleaned measurements. Single definition, used everywhere."""
    return pd.read_csv(ROOT / "data/processed/measurements_clean.csv")
```

WHAT ACTUALLY BREAKS

The console-only analysis is the failure that most often survives all the way into a submitted thesis, because it leaves no trace. The folder looks fine. There are scripts, there is data, there are figures. It is only when somebody asks for the specific number in Table 4 that anyone discovers it was produced by fourteen lines typed into a terminal in February.

13.7 Moving an existing project across

You don't have to convert everything at once.

1. Pick the number or figure you're least confident about.
2. Reconstruct the steps that made it, into a script.
3. Run the script from a clean state.
4. Compare with what's in your draft.

Step 4 is uncomfortable and it's the point. If the numbers differ, you have just found a real problem while you can still fix it quietly.

Then repeat for the next one. Two or three a week and a project converts inside a month.

DO THIS TODAY

Take the last thing you worked out in a console and write it as a script today, with a header comment and named constants. Run it in a fresh process. If the answer differs from what you had, that is the most useful thing you will learn this week.

14

One Command Rebuilds Everything

The goal of Part IV, stated as a target you can test against.

From raw data, one command produces every processed file, every figure, every table, and every number in your manuscript.

14.1 Why this is the right target

Because it's binary. Either the command works or it doesn't, and there's no room for "mostly" or "I think so".

It also collapses several problems at once. If one command rebuilds everything, then by construction there are no manual steps, no console-only analysis, no orphan figures, and no ambiguity about which script made what.

> *Delete your entire outputs folder and rebuild it. Whatever fails is the exact gap between the pipeline you think you have and the one you have. There is no other way to find that gap.*

14.2 The simplest version: a shell script

Start here. It takes ten minutes and it's most of the value.

```
#!/usr/bin/env bash
# run_all.sh -- rebuild everything from raw data.
# Usage:  bash run_all.sh

set -euo pipefail   # stop on the first error, not the last

echo "==> cleaning"
python code/01_clean.py

echo "==> fitting"
python code/02_fit.py

echo "==> figures"
python code/03_make_figures.py

echo "==> done"
```

That `set -euo pipefail` line matters more than the rest. Without it, a script that fails halfway is followed by the next one running on stale inputs, and you get a complete-looking set of outputs built on a broken step. With it, the run stops at the failure.

14.3 Making it honest

A build script that leaves old outputs in place can lie to you. Clear them first.

```
#!/usr/bin/env bash
set -euo pipefail

echo "==> clearing generated outputs"
rm -rf data/processed/* results/figures/* results/tables/*
mkdir -p data/processed results/figures results/tables

echo "==> cleaning";  python code/01_clean.py
echo "==> fitting";   python code/02_fit.py
echo "==> figures";   python code/03_make_figures.py
echo "==> done"
```

Note what is *not* deleted: `data/raw/` and `notes/`, the two irreplaceable artefacts. This is why the folder layout separates regenerable from irreplaceable, and it's why that separation has to be strict.

14.4 When to graduate to a build tool

A shell script reruns everything, every time. That's fine when the whole pipeline takes two minutes, and painful when the fitting step takes four hours.

A build tool tracks which outputs are out of date and reruns only what's needed.

`run_all.sh`	Under a few minutes end to end. Start here
`make`	Installed everywhere, language agnostic, awkward syntax
Snakemake	Python. Popular in bioinformatics; good for many similar files
targets	R. The standard choice in the R world
Nextflow, WDL	Cluster-scale pipelines. Almost certainly overkill

14.5 A Makefile, since it's everywhere

```
# Makefile -- run `make` to rebuild everything that is out of date.

CLEAN   = data/processed/measurements_clean.csv
PARAMS  = results/tables/fit_params.csv
FIGURES = results/figures/fig02_retention-vs-cycles.png

all: $(FIGURES)

$(CLEAN): code/01_clean.py data/raw/measurements.csv
        python code/01_clean.py
```

```
$(PARAMS): code/02_fit.py $(CLEAN)
        python code/02_fit.py

$(FIGURES): code/03_make_figures.py $(PARAMS) $(CLEAN)
        python code/03_make_figures.py

clean:
        rm -rf data/processed/*
        rm -rf results/figures/*
        rm -rf results/tables/*

.PHONY: all clean
```

Read a Make rule as: *output*, a colon, its dependencies, then an indented command that produces it. Make compares timestamps and reruns only what's stale.

So editing `02_fit.py` reruns the fit and the figures, and leaves the cleaning alone. Editing the raw data reruns everything.

The one gotcha: Make requires a real tab character at the start of each command line, not spaces. This has confused every person who has ever written their first Makefile, and the error message is unhelpful.

14.6 Declaring dependencies is the real work

Writing out what depends on what forces you to know your own pipeline, and that's most of the benefit. People routinely discover while doing it that two scripts both write the same file, or that a figure depends on a processed dataset nothing regenerates.

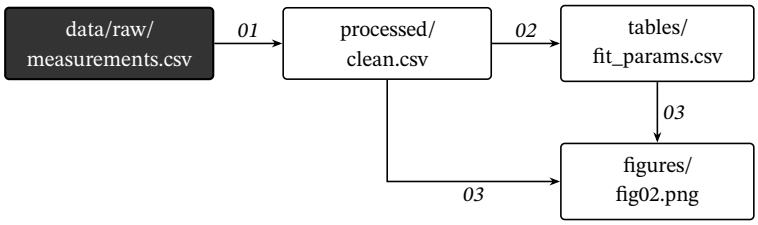

Figure 14.1: A dependency graph. Every arrow is a step that must be automated. An arrow you cannot draw is a manual step hiding in your pipeline.

14.7 The test, monthly

```
make clean && make        # or: bash run_all.sh
git status                # anything unexpectedly changed?
```

That second line is the clever part. If your outputs are reproducible and any of them are tracked, `git status` after a rebuild should show nothing. Anything that appears is a file whose content changed between two runs of the same code, which means something is non-deterministic. Chapter 16 covers why.

14.8 Long-running steps

If one step takes hours, don't let that stop you.

- **Cache the expensive output** as a processed file and let the build tool skip it when its inputs haven't changed. This is exactly what Make is for.
- **Keep a fast path** for development: a `--sample` flag that runs on 1% of the data, so you can test the pipeline end to end in a minute.
- **Log the long runs.** Write the start time, end time, and parameters to `results/logs/`, so you can tell which run produced which output.

WHAT ACTUALLY BREAKS

Almost every project has one step the author believes cannot be automated. Usually it's a GUI: an instrument's export dialog, a manual region selection, a fitting tool with buttons. Sometimes that's genuinely true. When it is, the answer isn't to give up on the build script, it's to write the manual step down as a numbered procedure in the README and have the script stop with a clear message telling the operator to do it. An honest break in the chain, documented, beats a chain that pretends to be complete.

DO THIS TODAY

Write `run_all.sh` today, even if it's three lines and even if one of them is an `echo` telling you to do something by hand. Then delete your figures folder and run it. What breaks is what you have learned.

15

The Environment Is Part of the Result

Your code doesn't say it needs pandas 2.1. It says `import pandas`.

One day pandas is version 4, a default has changed, and your analysis produces a different number without any error at all. That silent version is the one to worry about.

15.1 What the environment is

Everything outside your code that your code depends on.

Language version	Python 3.11 vs 3.9 vs 3.13
Libraries	pandas, numpy, scipy, and what they depend on
Their versions	The part everybody forgets
System tools	Compilers, `ffmpeg`, a LaTeX distribution
Operating system	Rarely matters, occasionally decisive

Recording it takes one command and it's the step almost everyone skips, because the environment is invisible while it works.

> *An analysis is code plus environment. Publishing one without the other is publishing half a method, and the half you left out is the half that changes on its own.*

15.2 Python

The one-command version. Create an isolated environment per project, and freeze it.

```
# once per project
python -m venv .venv
source .venv/bin/activate          # Windows: .venv\Scripts\activate
pip install pandas numpy matplotlib scipy

# record it
pip freeze > requirements.txt
git add requirements.txt && git commit -m "Record environment"
```

`requirements.txt` then pins exact versions:

```
matplotlib==3.8.2
numpy==1.26.3
pandas==2.1.4
scipy==1.11.4
```

Anyone, including you in two years, restores it with:

```
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
```

The virtual environment matters as much as the freeze. Without it, `pip freeze` records every package on your machine, including forty things from an unrelated project.

Add `.venv/` to `.gitignore`. It's hundreds of megabytes and it's rebuildable from the one file that matters.

15.3 conda, if you need compiled dependencies

```
conda create -n project-name python=3.11 pandas numpy matplotlib
conda activate project-name
conda env export --from-history > environment.yml
```

The `--from-history` flag records what you asked for rather than every transitive dependency with platform-specific build strings, which is what makes the file portable to a different operating system.

conda earns its keep when you need compiled libraries: GDAL, PyTorch with CUDA, R and Python in one environment.

15.4 R

```
install.packages("renv")
renv::init()        # once, in the project

# after installing packages
renv::snapshot()    # writes renv.lock

# to restore, elsewhere
renv::restore()
```

`renv.lock` records every package and version, and it goes in the repository. This is now the standard approach in R and it's better integrated than the Python equivalents.

15.5 Recording versions from inside the run

Belt and braces, and worth it. Have your build write down what it actually used.

```
# code/00_record_environment.py
import sys, platform, subprocess
from pathlib import Path
```

```
ROOT = Path(__file__).resolve().parents[1]
out = ROOT / "results" / "logs" / "environment.txt"
out.parent.mkdir(parents=True, exist_ok=True)

with out.open("w") as f:
    f.write(f"python   {sys.version}\n")
    f.write(f"platform {platform.platform()}\n")
    now = __import__("datetime").datetime.now()
    f.write(f"date     {now.isoformat()}\n\n")
    freeze = subprocess.run(
        [sys.executable, "-m", "pip", "freeze"],
        capture_output=True, text=True,
    )
    f.write(freeze.stdout)
```

In R, `sessionInfo()` does the same job in one call.

The difference between this and `requirements.txt` is that this records what ran, not what was declared. When the two disagree, that disagreement is the bug.

15.6 Containers, and whether you need one

A container packages the operating system too. Docker is the common tool.

Worth it when	Not worth it when
System-level dependencies are painful	`pip install -r` works
The analysis must run on a cluster	It runs on your laptop
You're publishing a tool others will use	You're submitting a thesis
The results are high-stakes and long-lived	Normal coursework

For most student projects a requirements file is enough, and a container is a week you could spend on the science. Learn it if your lab already uses one.

15.7 Pinning: how tight?

`pandas`	No pin. You will get a different version each install
`pandas>=2.0`	A floor. Fine while developing
`pandas==2.1.4`	Exact. What you want at submission

Work with loose pins if you like. **Freeze to exact versions when you submit, and commit that file with a tag.** That's the version somebody needs to reproduce the numbers in your thesis, and it's the one that matters.

WHAT ACTUALLY BREAKS

The version change that causes the most damage is never a crash. A crash tells you something is wrong on the day it happens. What actually hurts is a changed default: a function that used to drop missing values and now keeps them, a solver whose default tolerance tightened, a random generator that was reseeded differently. You get a number. It's plausible. It doesn't match your thesis, and nothing anywhere says why.

15.8 The five-minute setup

1. Create a virtual environment for this project.
2. Install what you need inside it.
3. `pip freeze > requirements.txt`, or `renv::snapshot()`.
4. Commit that file.
5. Add `.venv/` to `.gitignore`.

Five minutes, once, and it closes failure 5 from Chapter 1.

DO THIS TODAY

Do the five steps above today. Then, on a different machine or in a fresh environment, restore from your requirements file and run one script. What you're testing is not the file, it's whether the file is complete, and the answer is often no the first time.

16

Seeds and Determinism

If your analysis uses randomness anywhere, running it twice gives two answers unless you say otherwise.

The fix is one line. The chapter is about where randomness hides, which is more places than people expect.

16.1 Set the seed

```python
import numpy as np

SEED = 20260726                          # any fixed integer; record it
rng = np.random.default_rng(SEED)        # modern numpy: a generator object

sample = rng.choice(data, size=100, replace=False)
noise  = rng.normal(0, 0.1, size=len(data))
```

```r
set.seed(20260726)
sample <- sample(data, 100)
```

Now the same code gives the same answer every time, and somebody else running it gets your numbers rather than approximately your numbers.

> *Any script that uses randomness sets a seed at the top, as a named constant, and the seed is committed with the code. Without it, your results are not reproducible even by you.*

16.2 Prefer a generator object to a global seed

In Python, `np.random.seed()` sets a global state that any library you call can also change. `np.random.default_rng(SEED)` gives you an object nobody else touches.

FRAGILE

```python
np.random.seed(42)
# ... some library call that also uses np.random ...
x = np.random.normal(size=10)   # depends on what happened in between
```

REPRODUCIBLE

```
rng = np.random.default_rng(42)
# ... library calls cannot disturb this generator ...
x = rng.normal(size=10)          # depends only on the seed
```

The second is reproducible even if you add a library call in the middle later, which is exactly the sort of change that silently breaks the first.

16.3 Where randomness hides

The obvious places are sampling and simulation. The rest catch people out.

Train/test splits	`train_test_split(..., random_state=SEED)`
Cross-validation folds	Same, and it changes your reported score
Model initialisation	Neural network weights, k-means starts
Bootstrap and permutation	The whole method is resampling
Optimiser starting points	Many fitters start from a random guess
Shuffling	Batch order in training
Dictionary and set order	Python sets iterate in an order that varies between runs
Parallel execution	Results can arrive in a different order each run
Data loading	Some readers parallelise and return unordered rows

The last three aren't random number generators at all, and they produce the same symptom: run it twice, get two answers.

16.4 Non-determinism without randomness

Three sources worth knowing, because seeds don't fix them.

Set and dictionary iteration. In Python, iterating a `set` gives an order that can differ between runs. If you build a list of column names from a set and then index by position, your columns move. Sort it: `sorted(my_set)`.

Floating-point summation order. Adding the same numbers in a different order gives slightly different answers, because floating-point addition isn't associative. Usually this is a difference in the fifteenth decimal place and irrelevant. Occasionally, in an iterative solver near a threshold, it isn't.

Parallelism. Anything with `n_jobs=-1` or a thread pool can complete in a different order. If results are combined in completion order, they combine differently.

WHAT ACTUALLY BREAKS

The version of this that costs the most is the unseeded train/test split. A student reports 87% accuracy, a supervisor reruns it, and gets 84%. Neither number is wrong. The split

differed, and the honest reported figure was never a single number in the first place, it was a distribution over splits. That's a science problem revealed by a records problem, and it is much better found in a supervision meeting than in a viva.

16.5 Choosing and recording the seed

Any fixed integer works. Two habits worth having:

Pick it once and write it down. A date as 20260726 is as good as anything and tells you when the analysis was set up.

Never tune the seed. Trying seeds until the result looks good is a real and specific form of misconduct. If a conclusion depends on which seed you chose, the conclusion is that the result isn't stable, and that's worth reporting.

16.6 Reporting variability instead of hiding it

Better than a single seeded run, where the science allows it: run it many times and report the spread.

```
SEEDS = range(100)
scores = [evaluate(seed=s) for s in SEEDS]

print(f"accuracy {np.mean(scores):.3f} "
      f"+/- {np.std(scores):.3f} over {len(SEEDS)} seeds")
```

"0.862 $\pm$ 0.011 over 100 seeds" is a more honest and more useful number than "0.87", and a referee will prefer it. Fix the list of seeds so the summary itself is reproducible.

16.7 The two-run test

```
bash run_all.sh
cp -r results results_run1
bash run_all.sh
diff -r results results_run1
```

Any difference is non-determinism you haven't controlled. Silence means your pipeline is deterministic, and you can now claim that with evidence.

If some difference is unavoidable, floating-point noise in a solver, for instance, then say so in the README and state the tolerance.

DO THIS TODAY

Run the two-run test on your project today. Then search your code for every place randomness could enter, using the table above as the checklist, and confirm each one is seeded. The train/test split is the one most likely to be missing.

17

Notebooks: Where They Help and Where They Hurt

Jupyter and R Markdown are genuinely useful and they have one property that makes them dangerous for a pipeline.

This chapter is about using them without getting hurt, not about avoiding them.

17.1 The problem, in one screenshot's worth of words

A notebook's cells can be run in any order, and the notebook keeps the output of whatever ran last.

```
[3]  df = pd.read_csv("data.csv")
[7]  df = df[df.temp > 20]
[5]  print(len(df))            # 412
[9]  df = df[df.temp > 20]     # ran it twice, by accident
```

Those execution counts are the problem. The cells appear top to bottom; they ran 3, 7, 5, 9. The printed 412 was computed before a filter that appears above it. The filter has been applied twice.

Nothing here is an error. Every cell succeeded. The notebook looks completely normal, and the number it displays cannot be reproduced by running the notebook from top to bottom.

> *A notebook whose cells were not run top to bottom, in order, in one session, is not a record of anything. The execution counts tell you whether it is, and nobody looks at them.*

17.2 Where notebooks are the right tool

Exploring data	Excellent. Immediate feedback, plots inline
Developing an analysis	Excellent. Try things without rerunning everything
Teaching and demos	Excellent. Prose and code together
A report with figures	Good, especially R Markdown and Quarto
Sharing a worked example	Good

17.3 Where they hurt

Production pipeline	Hidden state, no ordering guarantee
Version control	`.ipynb` is JSON with embedded outputs; every diff is unreadable
Reuse	You cannot import a function from a notebook cleanly
Long-running steps	Nothing to resume from if the kernel dies
Automated running	Possible, and awkward

The version control point is worth expanding. A one-character change to a notebook produces a diff spanning hundreds of lines, because the file contains base64-encoded images and execution metadata. Your Git history becomes unreadable exactly where you most want to read it.

17.4 The rule that resolves it

Explore in a notebook. Move the result into a script.

This is Chapter 13 restated for notebooks. The notebook is scratch paper with better tooling, and scratch paper is not the record.

`notebooks/`	Exploration. Committed, but not part of the build
`code/`	Scripts. What `run_all.sh` calls

Both go in the repository. Only one is the pipeline.

17.5 If you must build with notebooks

Some fields do this and it can be done properly. Four requirements:

Restart and run all, always. Before committing, before believing a number, before showing anyone. In Jupyter: Kernel, then Restart Kernel and Run All Cells. If it fails, the notebook was lying to you.

Strip outputs before committing. This makes diffs readable.

```
pip install nbstripout
nbstripout --install        # installs a git filter for this repo
```

Now committed notebooks contain code and no outputs, so a diff shows the change you made.

Execute them from the command line in your build script, so the top-to-bottom run is enforced rather than hoped for.

```
jupyter nbconvert --to notebook --execute --inplace \
  notebooks/03_figures.ipynb
```

Or, better, with papermill, which lets you pass parameters in and writes the executed copy somewhere else:

```
papermill notebooks/03_figures.ipynb results/logs/03_figures_run.ipynb \
  -p seed 20260726
```

Put shared functions in a module, not in a cell. Import them into the notebook from `code/utils.py` so the notebook and the scripts use one definition.

17.6 The percent-script format

A middle path worth knowing about. Jupytext saves a notebook as a plain `.py` file with cell markers:

```
# %%
import pandas as pd
df = pd.read_csv("data/processed/clean.csv")

# %%
df.groupby("run").mean().plot()
```

That file is a valid Python script, runs top to bottom, diffs cleanly in Git, and opens as a notebook in VS Code, PyCharm, and Jupyter. For many projects it's the best of both, and it removes the whole hidden-state problem because there are no stored outputs at all.

17.7 R Markdown and Quarto are different

Worth saying plainly: R Markdown and Quarto do not have the hidden-state problem.

They render in a fresh session, top to bottom, every time. If it renders, it runs in order, from clean. That's a genuinely better design for reproducibility, and it's why the R community's practice around this is ahead of Python's.

```
quarto render analysis.qmd     # fresh session, top to bottom, or it fails
```

If your work is in R, using Quarto for the analysis document and calling it from your build script is a completely defensible pipeline.

WHAT ACTUALLY BREAKS

The notebook failure that reaches examiners most often is a figure whose notebook no longer runs. The plot is in the thesis, the notebook is in the repository, and running it top to bottom throws an error at cell four, because the working version depended on a variable defined in a cell that was later edited. The student is not being careless; the tool stored an output whose provenance it did not store.

DO THIS TODAY

Open your most important notebook and do Restart Kernel and Run All Cells. If it fails, or if any number changes, you have found a real problem. Then install `nbstripout` so your next commit produces a diff you can actually read.

Part V

Analysis and Outputs

18

Every Figure Made by Code

A figure adjusted by hand is a figure you cannot remake.

That matters at exactly one moment: eight months later, when a referee asks for one more data point on Figure 3.

18.1 The rule

Every figure in your thesis or paper is produced by a script, from processed data, with no manual step afterwards.

In the script	Never by hand
Axis limits and ticks	Dragging an axis in a GUI
Labels, units, legend	Typing a label into Illustrator
Colours and line styles	Recolouring in an image editor
Annotations and arrows	Adding an arrow in PowerPoint
Panel layout	Assembling panels in Inkscape
Output size and DPI	Cropping in Preview

> *If regenerating a figure requires you to remember a manual step, you will not remember it. The test is whether `make figures` produces the exact file that is in your manuscript.*

18.2 One script per figure

```
code/make_fig02.py  ->  results/figures/fig02_retention-vs-cycles.png
code/make_fig03.py  ->  results/figures/fig03_arrhenius-breakdown.png
```

Named for the number it has in the manuscript. When Figure 2 needs changing, you know which file to open, with no searching.

When the numbering changes during revision, rename both together the same afternoon. It stays a two-minute job only if you never let it accumulate.

18.3 A figure script that will still run

```
"""make_fig02.py -- retention against cycle count, by temperature.

Run:  python code/make_fig02.py
In:   data/processed/measurements_clean.csv
Out:  results/figures/fig02_retention-vs-cycles.png (300 dpi)
"""
from pathlib import Path
import pandas as pd
import matplotlib
matplotlib.use("Agg")            # no display needed; works on a cluster
import matplotlib.pyplot as plt

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "results" / "figures" / "fig02_retention-vs-cycles.png"
OUT.parent.mkdir(parents=True, exist_ok=True)

# One column width in the target journal, in inches.
FIGSIZE = (3.35, 2.6)
DPI = 300
TEMPS = [25, 35, 45]
MARKERS = {25: "o", 35: "s", 45: "^"}     # distinguishable in greyscale

df = pd.read_csv(ROOT / "data/processed/measurements_clean.csv")

fig, ax = plt.subplots(figsize=FIGSIZE)
for t in TEMPS:
    sub = df[df.temp_c == t]
    ax.plot(sub.cycles, sub.retention,
            marker=MARKERS[t], markersize=3, linewidth=1,
            label=f"{t} řC")

ax.set_xlabel("Cycle number")
ax.set_ylabel("Capacity retention (%)")
ax.set_xlim(0, 500)
ax.set_ylim(60, 100)
ax.legend(frameon=False, fontsize=8)
fig.tight_layout()
fig.savefig(OUT, dpi=DPI)
print(f"wrote {OUT.relative_to(ROOT)}")
```

Six things in there are deliberate. `matplotlib.use("Agg")` so it runs without a display. An explicit figure size in the journal's column width, so nothing is rescaled afterwards. Markers as well as colours, so it survives greyscale printing. Explicit axis limits, so the figure doesn't change shape when the data does. `mkdir` so a fresh clone works. And a printed confirmation of what it wrote.

18.4 Size it for the page, not the screen

Set the figure size in inches to the width it will be printed at, and don't scale it in LaTeX.

Single column	About 3.3 in (8.5 cm) in most journals
Double column	About 6.9 in (17.5 cm)
Thesis, one column	Your text width; check the class file

Scaling with `[width=0.8\textwidth]` shrinks the fonts along with everything else, which is why so many published figures have unreadable axis labels. Set the size once, in the script, and include at natural size.

18.5 Vector or raster?

PDF, SVG, EPS	Vector. Line plots, schematics. Sharp at any size
PNG	Raster. Use for images, heatmaps, and scatter plots with tens of thousands of points
300 dpi	Minimum for raster in print. 600 for line art
Never JPEG	Lossy compression puts artefacts around every line and letter

Save both if you like: PDF for the manuscript, PNG for slides and for looking at quickly.

18.6 When a figure genuinely needs hand assembly

Sometimes it does. A multi-panel figure combining a photograph, a schematic drawn in Inkscape, and two plots is a real case.

Handle it honestly:

1. Generate every plot panel by script, at final size.
2. Keep the editable source of the hand-drawn parts in `docs/figure-sources/`, committed.
3. Write the assembly steps in `docs/figure-sources/README.md`, numbered, so somebody could repeat them.
4. Have the build script produce the panels and stop with a message saying assembly is manual.

That's a documented break in the chain, which is a completely different thing from an undocumented one.

WHAT ACTUALLY BREAKS

The most common irreproducible figure is not hand-drawn at all. It's a plot that was generated by a script, then opened once to fix a label position or nudge a legend, and re-saved over the top. The script still exists and still runs, and it produces a subtly different file. Nobody can tell which one is in the paper, and the person least able to tell is the author.

18.7 Consistency across figures

Put the shared styling in one place so every figure matches.

```
# code/plotstyle.py
import matplotlib.pyplot as plt

def apply_style():
    plt.rcParams.update({
        "font.size": 8,
        "axes.labelsize": 8,
        "axes.spines.top": False,
```

```
        "axes.spines.right": False,
        "legend.frameon": False,
        "savefig.bbox": "tight",
    })

TEMP_COLOURS = {25: "#4d7c0f", 35: "#b45309", 45: "#9f1239"}
TEMP_MARKERS = {25: "o", 35: "s", 45: "^"}
```

Import it in every figure script. Now 35 °C is the same colour and the same marker in all eight figures, and changing that decision is a one-line edit rather than eight.

18.8 The test

```
rm -rf results/figures/*
bash run_all.sh
```

Then compare what came out against what's in your manuscript. Any figure that didn't regenerate, or regenerated differently, is one you cannot defend a revision on.

DO THIS TODAY

Delete one figure from `results/figures/` and regenerate it with its script. Then open it next to the version in your draft and look carefully. If they differ, the difference is a manual step you had forgotten, and finding it now is worth an hour.

19

Numbers That Trace Back

Every number in your manuscript should be traceable to the script that produced it.

The alternative is retyping, and retyping is where transcription errors live.

19.1 The problem

You run the fit, see `slope = -0.04117`, and type −0.041 into your draft.

Three things can now go wrong. You mistype it. You update the analysis and forget to update the text. Or a co-author asks where the number came from and the answer is “the console, in March”.

The third is the one that matters, because it’s the one you cannot fix later.

> *A number in your manuscript that was typed by hand is a number nobody can verify. If your analysis changes and the text doesn’t, nothing anywhere will tell you.*

19.2 Level 1: write the numbers to a file

The minimum, and it takes one line.

```
results = {
    "slope_per_cycle": slope,
    "intercept_pct": intercept,
    "n_runs": len(per_run),
    "n_excluded": n_excluded,
    "retention_25c_500cyc": ret25,
    "retention_45c_500cyc": ret45,
}
pd.Series(results).to_csv(ROOT / "results/tables/key_numbers.csv")
```

Now there is one file holding every number that appears in your text. When you write the manuscript you copy from that file, and when the analysis changes you know exactly which file to re-read.

Not automatic, and it converts “where did this come from” from an unanswerable question into a lookup.

19.3 Level 2: let the document read the numbers

If you write in LaTeX, have your analysis emit macros.

```
def tex_macro(name, value):
    command = f"\\newcommand{{\\{name}}}"
```

```
    return f"{command}{{{value}}}\n"

with open(ROOT / "results/tables/numbers.tex", "w") as f:
    f.write(tex_macro(
        "slopePerCycle", f"{slope:.4f}"
    ))
    f.write(tex_macro(
        "nRuns", len(per_run)
    ))
    f.write(tex_macro(
        "nExcluded", n_excluded
    ))
    f.write(tex_macro(
        "retTwentyFive", f"{ret25:.1f}"
    ))
    f.write(tex_macro(
        "retFortyFive", f"{ret45:.1f}"
    ))
```

```
% in the manuscript preamble
\input{../results/tables/numbers.tex}

% in the text
Retention after 500 cycles fell from \retTwentyFive\% at
25\,\dg C to \retFortyFive\% at 45\,\dg C, across \nRuns\ runs
(\nExcluded\ excluded).
```

Now rerunning the analysis and recompiling the document updates the prose. The numbers in your text cannot disagree with your results, because they are the same numbers.

This is the single highest-return technique in the chapter and almost nobody does it.

19.4 Level 3: literate documents

R Markdown and Quarto do this natively, computing the values as the document renders.

```
Retention fell from `r round(ret25, 1)`% at 25 řC to
`r round(ret45, 1)`% at 45 řC across `r n_runs` runs.
```

Python's equivalents are Quarto with a Python kernel, or Jupyter Book.

The trade-off: the document and the analysis become one thing, which is excellent for a report and awkward when the analysis takes four hours and you want to fix a typo. The common compromise is level 2, where the analysis runs separately and writes a small file the document reads.

19.5 Tables, generated

The same argument applies, more strongly, because tables have more numbers to mistype.

```
summary = per_run.groupby("temp_c").agg(
    n=("retention", "size"),
    mean_retention=("retention", "mean"),
    sd_retention=("retention", "std"),
```

```
).round(2)

summary.to_csv(ROOT / "results/tables/tab01_summary.csv")
summary.to_latex(ROOT / "results/tables/tab01_summary.tex",
                 float_format="%.2f", bold_rows=False)
```

Then `\input` the `.tex` file in your manuscript. A referee asks you to add a column, and it's a one-line change in one place rather than a retyping exercise.

19.6 Rounding, once, at the end

Round for display, never in the pipeline.

FRAGILE

```
mean_temp = round(df.temp_c.mean(), 1)      # rounded here
correction = mean_temp * COEFFICIENT        # and the error propagates
```

REPRODUCIBLE

```
mean_temp = df.temp_c.mean()                # full precision throughout
correction = mean_temp * COEFFICIENT
print(f"mean temperature {mean_temp:.1f} řC")   # rounded for display
    ↪ only
```

And report only the precision your measurement supports. Writing `87.4213%` from six samples claims a precision you do not have, and a referee will say so.

19.7 The consistency check before submitting

Twenty minutes, and it catches real errors.

1. List every number in the manuscript: text, tables, figure captions, and the abstract.
2. For each, find it in `results/`.
3. Check every sample size reconciles: collected minus excluded equals analysed.
4. Check the abstract's numbers match the results section's.

Most first drafts fail at least one of the four, and step 4 is the one that fails most, because the abstract was written first and the analysis moved underneath it.

WHAT ACTUALLY BREAKS

The mismatch that turns up most often at proof stage is between the abstract and a table: the abstract says 87% and the table says 86.4%, because the abstract was written in April and the analysis was rerun in June. Neither number is wrong. The paper is simply inconsistent, and a careful reader will find it and wonder what else moved.

19.8 Where this pays off

The revision. A referee asks for one additional control, you add it, rerun, recompile, and every affected number in the manuscript updates itself.

The alternative is finding all of them by hand, under a deadline, in a document you last read six months ago, and missing one.

DO THIS TODAY

Write your key numbers to a file today, from the script that computes them. If you use LaTeX, take the extra ten minutes to emit macros and replace three hand-typed numbers in your draft with them. You will not go back.

20

Checking Analysis Code

You are not going to write a test suite, and you should not need a software-engineering course to catch the errors that matter.

This chapter is the small subset of testing that pays for itself in a student project.

20.1 The errors that actually happen

Not subtle algorithmic bugs. These five, over and over.

Silent row loss	A merge or filter drops rows and nothing says so
Wrong units	Metres and millimetres, Celsius and Kelvin, radians and degrees
Off-by-one	A window, an index, a fencepost
Misaligned join	Two tables merged on a key that isn't unique
Missing values	`NaN` propagating, or being silently dropped by a mean

Every one of these produces a plausible number. That's what makes them dangerous: nothing crashes, and the output looks like an answer.

> *The bug that hurts is not the one that crashes. It's the one that returns a number you believe. Your checks should target plausibility, not correctness.*

20.2 Assertions: the whole technique in one line

An assertion says "this must be true here". If it isn't, the script stops.

```
raw = pd.read_csv(ROOT / "data/raw/measurements.csv")
assert len(raw) == 4812, f"expected 4812 rows, got {len(raw)}"

clean = clean_measurements(raw)
assert len(clean) == len(raw) - 4, "unexpected number of rows dropped"
assert clean.temp_c.between(-40, 120).all(), "temperature out of range"
assert clean.retention.notna().all(), "missing retention values"
assert clean.run_id.is_unique, "duplicate run ids after cleaning"
```

Five lines. They catch four of the five errors in the table above, and they took ninety seconds to write.

In R:

```
stopifnot(
  nrow(clean) == nrow(raw) - 4,
  all(clean$temp_c > -40 & clean$temp_c < 120),
  !any(is.na(clean$retention))
)
```

> *Write an assertion every time you find yourself thinking "it should have about this many rows". That thought is a check you have already performed mentally; spending ten seconds to make it permanent is the whole technique.*

20.3 Where to put them

At every boundary where data changes shape.

After loading	Row count, expected columns, no all-null columns
After filtering	How many rows went, and is that the number you expected
After a merge	Row count did not grow. This one catches the duplicate-key bug
After a transform	Values in a physically possible range
Before writing out	No missing values where there should be none

The merge check deserves emphasis. A join on a non-unique key silently multiplies rows, so 400 rows become 1,600 and every subsequent mean is wrong. It's one of the most common serious errors in student data work and one assertion catches it:

```
before = len(df)
df = df.merge(lookup, on="run_id", how="left", validate="many_to_one")
assert len(df) == before, f"merge changed row count: {before} -> {len(df)}"
```

That `validate="many_to_one"` argument makes pandas itself refuse the merge if the key isn't unique on the right. Use it every time.

20.4 Sanity-check the answer, not just the data

Before believing a result, check it against something you know independently.

Order of magnitude	Is the answer roughly what physics or prior work suggests?
Sign	Should this be negative?
Units	Does the number make sense in the units you think it's in?
Limiting case	Set a parameter to zero. Does the answer go where it must?
A known answer	Run your code on a case with a published result
Do it by hand	One data point, on paper, with a calculator

That last one is undervalued. Computing one row of your analysis by hand takes fifteen minutes and catches unit errors, transposed indices, and misread column meanings in a way no automated check will.

20.5 The limiting-case check, concretely

```
# A degradation model must return full retention at zero cycles,
# whatever the fitted parameters are.
assert abs(model(cycles=0, **params) - 100.0) < 1e-6

# And it must be monotonically decreasing in cycles.
r = [model(cycles=c, **params) for c in range(0, 501, 50)]
assert all(a >= b for a, b in zip(r, r[1:])), "retention increased"
```

Two checks that encode physics rather than arithmetic. If either fails, the model is wrong regardless of how good the fit statistics look.

20.6 Testing a function properly, when it's worth it

For a function doing real logic, write a handful of cases with known answers.

```
# code/test_utils.py   -- run with:  pytest code/
from utils import fahrenheit_to_celsius as f2c

def test_freezing():
    assert f2c(32) == 0

def test_boiling():
    assert f2c(212) == 100

def test_negative():
    assert f2c(-40) == -40      # the crossover point

def test_array():
    import numpy as np
    np.testing.assert_allclose(f2c(np.array([32, 212])), [0, 100])
```

```
pip install pytest
pytest code/
```

Four tests, two minutes. Worth it for any function you'll rely on more than once, and not worth it for a one-off plotting script.

20.7 The regression check

Once your pipeline produces the right answer, record it. Then you'll know if it changes.

```
bash run_all.sh
cp results/tables/key_numbers.csv results/expected_numbers.csv
git add results/expected_numbers.csv
```

```
git commit -m "Record expected results as of 2026-03-19"
```

```
# later, after any change
diff results/tables/key_numbers.csv results/expected_numbers.csv
```

If the diff is empty, your change didn't alter the results. If it isn't, you find out immediately and deliberately rather than in eight months.

This is the single most useful check in the chapter for a project that's already working, because it costs one file and catches every accidental change.

WHAT ACTUALLY BREAKS

The bug found latest is nearly always a unit error, and the reason is that units are the one thing no tool checks. A temperature in Fahrenheit where Kelvin was expected produces numbers that are wrong by a factor and a constant, which still fit, still plot, and still look like results. The only defences are naming variables with their units, as in `temp_c` rather than `temp`, and asserting ranges.

DO THIS TODAY

Add three assertions to your cleaning script today: the row count after loading, the row count after filtering, and a physically possible range on your main variable. Then record your current results as `expected_numbers.csv` and commit it.

21

Documenting for Your Future Self

Your first collaborator is you, in six months, with no memory of any of this.

That person cannot be emailed, cannot ask a question, and is the reader every comment in this chapter is written for.

21.1 What to document, and what not to

Document	Don't bother
Why, not what	`# increment i by one`
Non-obvious choices	Anything the code says plainly
Units and conventions	Restating a well-named function's name
Where a number came from	A comment that will drift out of date
Known limitations	Aspirational comments about future features
The pipeline order	

FRAGILE

```
# read the csv
df = pd.read_csv("data.csv")

# filter
df = df[df.v < 4.2]

# loop over rows
for i in range(len(df)):
    ...
```

REPRODUCIBLE

```
df = pd.read_csv(ROOT / "data/processed/measurements_clean.csv")

# Cells above 4.2 V were still in the constant-current phase, so their
# retention figures are not comparable. Threshold from the cell
# datasheet, section 4.2; see notes/2026-03-19.md.
df = df[df.voltage_v < MAX_VOLTAGE_V]
```

> *Comment the decisions, not the mechanics. If a comment could be deleted without losing information, it was describing the code rather than the reasoning.*

```
"""02_fit.py -- fit the two-mechanism degradation model.

Run:   python code/02_fit.py
In:    data/processed/measurements_clean.csv
Out:   results/tables/fit_params.csv
       results/tables/numbers.tex
Notes: Excludes runs above 4.2 V (still in CC phase). Fits on runs
       1-24; runs 25-31 held out for validation.
Time:  about 40 seconds.
"""
```

```
def arrhenius_rate(temp_c, ea_ev, prefactor):
    """Degradation rate from the Arrhenius relation.

    temp_c:    temperature in degrees Celsius (converted to K inside)
    ea_ev:     activation energy in electron volts
    prefactor: rate at infinite temperature, in % per cycle
    returns:   rate in % capacity loss per cycle
    """
```

WHAT ACTUALLY BREAKS

The comment that turns out to matter is almost never an explanation of the code. It's a sentence like "the datasheet says 4.2 V but the measured plateau is 4.18, so this threshold is deliberately slightly low". Nobody can reconstruct that from the code, nobody remembers it after a term, and it is exactly what a referee or an examiner asks about.

1.
2.
3.

4.
5.

DO THIS TODAY

Add a four-line header to every script in your current project. It takes about twenty minutes for a typical project and it is the highest-value twenty minutes of documentation you will ever do. Then find one magic number in your code and write the sentence explaining it.

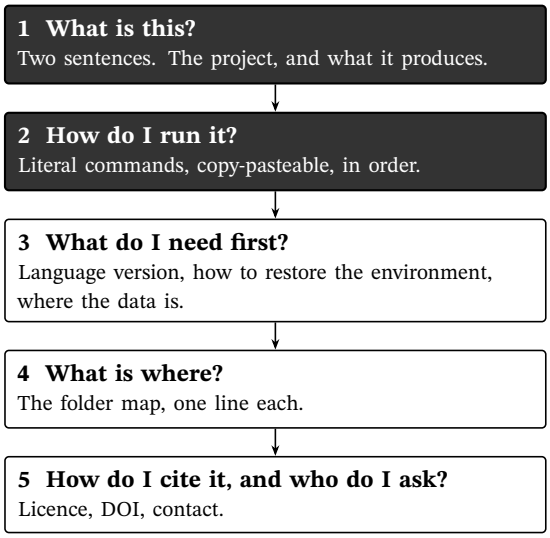

Figure 22.1: The five questions, in priority order. A reader who gets answers to the first three can already use the project.

```
# Perovskite thermal cycling analysis

Analysis code for "Grain-boundary ingress dominates perovskite
degradation above 35 C" (MSc thesis, 2026). Takes cycling data from
24 cells at four temperatures and produces the two-mechanism model
fit and all five figures in the thesis.

## Quick start

    git clone https://github.com/you/perovskite-cycling.git
```

```
    cd perovskite-cycling
    python -m venv .venv && source .venv/bin/activate
    pip install -r requirements.txt
    bash run_all.sh

Takes about 6 minutes. Writes figures to results/figures/ and the
key numbers to results/tables/key_numbers.csv.

## Requirements

- Python 3.11 (3.9+ should work; only tested on 3.11)
- Packages: see requirements.txt
- No GPU, no cluster. Runs on a laptop.

## Data

Raw cycling data is in data/raw/ (12 MB, included).
Provenance, instrument settings, and checksums: data/raw/README.md

Raw data is read-only by design. All cleaning happens in
code/01_clean.py and writes to data/processed/.

## Layout

    code/           analysis scripts, numbered in run order
    data/raw/       instrument output. READ-ONLY, never edited
    data/processed/ generated. safe to delete
    results/        figures, tables, logs. all generated
    notes/          lab notebook and the decisions log
    docs/           collection protocol, calibration certificates

## Pipeline

    01_clean.py        raw -> processed. Excludes 4 of 28 runs;
                       rules and reasons in notes/decisions.md
    02_fit.py          processed -> fit parameters
    03_make_figures.py -> results/figures/fig01..fig05

## Known limitations

- All cells from one manufacturing lot; lot-to-lot variation is
  not captured.
- The 35 C crossover is a fitted parameter, not measured directly.

## Citing

If you use this, please cite the thesis and the archived code:
[DOI once deposited]. Code is MIT licensed; data is CC BY 4.0.

## Contact

Gaurav Tiwari, gaurav@example.ac.uk
```

> *Write the quick start as literal commands, then test them by following your own instructions in a fresh clone. Every README that fails, fails here, and the author is the last person able to notice.*

```
cd /tmp
git clone https://github.com/you/project.git test-clone
cd test-clone
# now follow your own README, exactly, changing nothing
```

```
# Perovskite thermal cycling

MSc project on capacity fade above 35 C.

## Run
    bash run_all.sh
```

WHAT ACTUALLY BREAKS

The instruction most often missing from a student README is the one the author stopped seeing as an instruction. Activating the virtual environment. Being on the university VPN to reach the data store. Running from the project root rather than from inside `code/`. Each is invisible to the person who does it by reflex, and each stops a stranger dead in the first two minutes.

```
cff-version: 1.2.0
title: Perovskite thermal cycling analysis
authors:
  - family-names: Tiwari
    given-names: Gaurav
version: 1.0.0
date-released: 2026-09-01
license: MIT
```

DO THIS TODAY

Write the five-question README today, then clone your project into a temporary folder and follow it exactly, changing nothing. Fix whatever fails. That hour is what separates code somebody can use from code somebody gives up on.

> *MIT for code, CC BY 4.0 for data, unless your institution or a funder says otherwise. Check first: some universities claim ownership of student work, and some funders mandate a specific licence.*

1.
2.
3.
4.
5.

```
## Data availability

The interview transcripts cannot be shared publicly under the terms of
ethics approval 2026/117. What is available:

- All analysis code (this repository, DOI: 10.5281/zenodo.XXXXXXX)
- De-identified summary statistics (data/processed/summary.csv)
- A synthetic dataset with matched structure, so the pipeline runs
  end to end (data/synthetic/, generated by code/00_make_synthetic.py)

Requests for access to the underlying data may be made to the data
controller at research-governance@example.ac.uk.
```

```
git tag -a v1.0.0 -m "Version accompanying the thesis, September 2026"
git push --tags
```

WHAT ACTUALLY BREAKS

The archiving step gets skipped because it happens at the exact moment a project ends, when everybody involved is finished with it. It takes twenty minutes and it is the difference between work that stays findable and work that exists on a laptop that

gets wiped when you hand it back. The people who do it are not more organised; they just did it on the day they submitted rather than intending to do it later.

DO THIS TODAY

Add a `LICENSE` file today, MIT for code, and check whether your institution has a rule about it. Then connect your repository to Zenodo, so that when you tag a release the DOI appears without you having to remember anything.

> *A project that only works while you are available to explain it is not finished. It is a dependency on you, and you are leaving.*

```
# notes/known-issues.md

## Things that don't work

- 03_make_figures.py fails on Windows: the path separator in
  save_panel() is hard-coded. Two-line fix, never got to it.

- The bootstrap in 02_fit.py takes 40 minutes because it refits
  from scratch each iteration. Could cache the design matrix.

## Things I'm not sure about

- The 35 C crossover is a fitted parameter and I never validated it
  independently. If it moves when new data is added, that's expected,
  and it would weaken the two-mechanism claim.

- Runs 8 and 19 have slightly odd early-cycle behaviour. I kept them
  because no exclusion rule justified dropping them, but if anyone
  finds an instrument reason, they're the ones to look at.

## Things I'd do differently

- Should have recorded ambient humidity. It probably matters above
  35 C and we have no measurement of it.
```

```
# Start here

This project cycles 24 perovskite cells at four temperatures and asks
whether the standard Arrhenius lifetime model holds above 35 C. It
doesn't, and the two-mechanism fit in 02_fit.py is the main result.
```

```
Read in this order:
  1. docs/protocol.md      how the data was collected
  2. notes/decisions.md    every choice I made, and why
  3. code/01_clean.py      the exclusions live here
  4. notes/known-issues.md what's broken and what I'm unsure about

Then run `bash run_all.sh` and compare results/tables/key_numbers.csv
against results/expected_numbers.csv. They should match exactly.

The thing to be careful about: temperature is in Celsius everywhere
except inside arrhenius_rate(), which converts to Kelvin internally.
I got that wrong twice.
```

WHAT ACTUALLY BREAKS

The handover that goes wrong is rarely missing files. It's a project where everything is present and nothing is explained: the code runs, the data is there, and the new student cannot tell which of four scripts is authoritative, or why 4 of 28 runs were dropped. They then spend six weeks reconstructing decisions that took the previous student ten seconds each to make, and often reach different ones.

-
-
-
-

DO THIS TODAY

Write `notes/known-issues.md` today, before the end of the project, while you still remember what's broken and what you're unsure about. Then put your permanent email in the README. Ten minutes for both, and they are what make the work survive you.

```
project-name/
  README.md
  LICENSE
  CITATION.cff
  requirements.txt        or environment.yml / renv.lock
  .gitignore
  run_all.sh              or Makefile
  backup.sh
  data/
    raw/
      README.md           provenance: source, date, settings, columns
      checksums.txt
    processed/
      .gitkeep
    external/
      README.md           where reference data came from
  code/
    00_record_environment.py
    01_clean.py
    02_fit.py
    03_make_figures.py
    utils.py
    plotstyle.py
  results/
    figures/.gitkeep
    tables/.gitkeep
    logs/.gitkeep
  notes/
    decisions.md
    known-issues.md
  docs/
    protocol.md
```

```
#!/usr/bin/env bash
# new-project.sh NAME -- create a reproducible project skeleton.
set -euo pipefail

NAME="${1:?usage: new-project.sh PROJECT-NAME}"
mkdir -p "$NAME"/data/{raw,processed,external}
mkdir -p "$NAME"/results/{figures,tables,logs}
mkdir -p "$NAME"/{code,notes,docs}
cd "$NAME"

touch data/processed/.gitkeep results/{figures,tables,logs}/.gitkeep

cat > README.md <<'EOF'
# PROJECT NAME

One or two sentences: what this is and what it produces.

## Quick start

    python -m venv .venv && source .venv/bin/activate
    pip install -r requirements.txt
    bash run_all.sh

## Layout

    code/           analysis scripts, numbered in run order
    data/raw/       READ-ONLY. never edited
    data/processed/ generated. safe to delete
    results/        figures, tables, logs. all generated
    notes/          lab notebook and decisions log
    docs/           protocol and reference material
EOF

cat > .gitignore <<'EOF'
data/raw/*
!data/raw/README.md
!data/raw/checksums.txt
data/processed/*
!data/processed/.gitkeep
results/figures/*
results/tables/*
results/logs/*
!results/**/.gitkeep

.env
*.key
*.pem
credentials.json

__pycache__/
*.pyc
.venv/
.Rhistory
.RData
.Rproj.user/
.ipynb_checkpoints/
```

```
.DS_Store
Thumbs.db
.vscode/
.idea/
EOF

cat > run_all.sh <<'EOF'
#!/usr/bin/env bash
set -euo pipefail

echo "==> clearing generated outputs"
rm -rf data/processed/* results/figures/* results/tables/*
mkdir -p data/processed results/figures results/tables

echo "==> environment"; python code/00_record_environment.py
echo "==> cleaning";    python code/01_clean.py
echo "==> fitting";     python code/02_fit.py
echo "==> figures";     python code/03_make_figures.py
echo "==> done"
EOF
chmod +x run_all.sh

cat > backup.sh <<'EOF'
#!/usr/bin/env bash
# Mirror the irreplaceable artefacts. No --delete, deliberately.
set -euo pipefail
DEST="${1:?usage: backup.sh /path/to/backup}"
rsync -av data/raw/ "$DEST/data/raw/"
rsync -av notes/    "$DEST/notes/"
EOF
chmod +x backup.sh

cat > data/raw/README.md <<'EOF'
# Raw data

READ-ONLY. Never edited. All cleaning happens in code/ and writes to
data/processed/.

## FILENAME.csv
Source:
Collected:
Settings:
Calibration:
Columns:
Rows:
Notes:
EOF

cat > notes/decisions.md <<'EOF'
# Decisions

| Date | Decision | Why |
|------|----------|-----|
EOF
```

```
cat > notes/known-issues.md <<'EOF'
# Known issues

## Things that don't work

## Things I'm not sure about

## Things I'd do differently
EOF

cat > code/utils.py <<'EOF'
"""Shared helpers. Import these; do not copy-paste between scripts."""
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
EOF

git init -q && git add -A && git commit -qm "Project skeleton"
echo "created $NAME (git initialised, .gitignore in place)"
```

```
chmod +x new-project.sh
./new-project.sh 2026_msc-thesis_perovskite-cycling
```

```
code/01_clean.R
code/02_fit.R
code/03_make_figures.R
code/utils.R
renv.lock               # instead of requirements.txt
project-name.Rproj      # so RStudio sets the working directory
```

```
# in each script, instead of pathlib
library(here)
df <- read.csv(here("data", "processed", "clean.csv"))
```

```
Rscript code/01_clean.R
Rscript code/02_fit.R
Rscript code/03_make_figures.R
```

1.

2.
3.
4.
5.

6.

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
git config --global pull.rebase false
```

```
# write .gitignore FIRST, then:
git init
git add .
git commit -m "Initial commit: project skeleton"

# connect a remote
git remote add origin https://github.com/you/project.git
git push -u origin main
```

```
git status                          # what changed?
git diff                            # what exactly?
git add FILE                        # stage one file
git add -A                          # stage everything
git diff --staged                   # read before committing
git commit -m "Message"             # save a snapshot
git push                            # send it off-site
```

```
git commit -a -m "Message" && git push
```

```
git log --oneline                    # compact
git log --oneline -10                # last ten
git log --oneline --stat             # with files changed
git log --grep="run 47"              # search messages
git log -p code/02_fit.py            # one file, with diffs
git log --since="2026-03-01"         # by date
git show HEAD                        # the last commit in full
git show HEAD~3:code/02_fit.py       # a file as it was 3 commits ago
git blame code/02_fit.py             # who changed each line, and when
```

```
git switch -c branch-name            # create and switch
git switch main                      # switch back
git branch                           # list
git merge branch-name                # bring work into main
git branch -d branch-name            # delete after merging
git merge --abort                    # back out of a messy merge

git tag -a v1.0.0 -m "Thesis version, September 2026"
git push --tags
git tag                              # list tags
```

```
git remote -v                       # where does this push to?
git push                            # send commits
git pull                            # fetch and merge
git fetch                           # fetch without merging
git clone URL                       # first download
git clone URL --depth 1             # latest state only, faster
```

```
git rev-list --objects --all \
  | git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize)
    ↪ %(rest)' \
  | awk '$1=="blob" {print $3, $4}' | sort -rn | head -10
```

```
git config --global alias.s "status -s"
git config --global alias.l "log --oneline --graph -20"
git config --global alias.d "diff --staged"
git config --global alias.save "!git add -A && git commit -m"
```

□

□
□
□
□
□
□
□

□
□
□

□
□
□

□
□
□
□
□
□
□
□

□
□
□
□
□

□
□
□
□
□

□
□
□
□
□
□

□
□
□
□

□

□
□
□
□
□
□
□
□

□
□

- [ ]
- [ ]
- [ ]

- [ ]
- [ ]

- [ ]
- [ ]

- [ ]
- [ ]
- [ ]
- [ ]

- [ ]
- [ ]
- [ ]

- [ ]
- [ ]

- [ ]

1.
2.
3.
4.
5.
6.

```
# [Project title]

[Two sentences: what this project does and what it produces. Name the
thesis or paper it belongs to.]

## Quick start

    git clone [URL]
    cd [project]
    python -m venv .venv && source .venv/bin/activate
    pip install -r requirements.txt
    bash run_all.sh

Takes about [N] minutes. Writes figures to results/figures/ and the key
numbers to results/tables/key_numbers.csv.

## Requirements

- [Language] [version] ([which versions you actually tested])
- Packages: see requirements.txt
- [Any system dependency: a compiler, ffmpeg, a LaTeX distribution]
- [Hardware: laptop / GPU / cluster]

## Data

[Where it is. If included, how big. If not, how to fetch it.]

Provenance, instrument settings, and checksums: data/raw/README.md

Raw data is read-only by design. All cleaning happens in
code/01_clean.py and writes to data/processed/.

## Layout

    code/           analysis scripts, numbered in run order
    data/raw/       READ-ONLY. never edited
    data/processed/ generated. safe to delete
    results/        figures, tables, logs. all generated
    notes/          lab notebook and decisions log
```

```
    docs/           protocol and reference material

## Pipeline

    01_clean.py         raw -> processed. [What it excludes, and where the
                        rules are written down]
    02_fit.py           processed -> [what]
    03_make_figures.py -> results/figures/fig01..figNN

## Known limitations

- [Something honest about scope]
- [Something honest about a choice you could not validate]

## Citing

Please cite [thesis/paper] and the archived code: [DOI].
Code is [MIT] licensed; data is [CC BY 4.0].

## Contact

[Name], [an email that will still work in three years]
```

```
# Raw data

READ-ONLY. Never edited. All corrections happen in code/ and are written
to data/processed/.

## [filename]
Source:      [instrument, model, location / survey platform / URL]
Collected:   [date range, times, by whom]
Settings:    [every setting that affects the numbers]
Calibration: [date, and where the certificate is]
Columns:     [name: meaning and unit, one per line]
Rows:        [count]
Notes:       [anything odd. Known problems. Corrections applied later
             in code, and where]
Checksum:    [sha256, or see checksums.txt]
```

```
All data and analysis code are available at [DOI]. Raw cycling data,
processed datasets, the full analysis pipeline, and the environment
specification are included. Figures in this paper are regenerated by
`bash run_all.sh`.
```

```
The [interview transcripts / patient records] analysed here cannot be
shared publicly under the terms of ethics approval [ref]. The following
are available at [DOI]: the complete analysis code, de-identified
summary statistics, and a synthetic dataset with matched structure that
allows the pipeline to be run end to end. Requests for access to the
underlying data may be made to the data controller at [address].
```

```
Analysis code is archived at [DOI] (version [v1.0.0]) and developed at
[repository URL]. The environment is specified in requirements.txt;
`bash run_all.sh` reproduces every figure and table in this paper from
the raw data.
```

```
cff-version: 1.2.0
title: [Project title]
message: If you use this software, please cite it as below.
type: software
authors:
  - family-names: [Surname]
    given-names: [Given names]
    orcid: https://orcid.org/0000-0000-0000-0000
    affiliation: [Institution]
version: 1.0.0
date-released: 2026-09-01
license: MIT
doi: 10.5281/zenodo.XXXXXXX
repository-code: https://github.com/you/project
```

```
# Known issues

## Things that don't work
- [What breaks, on what platform, and how big the fix looks]

## Things I'm not sure about
- [A result that might not be robust, and what would test it]
- [A data point I kept or dropped and could argue either way]

## Things I'd do differently
- [A measurement I wish we had taken]
```

```
# Decisions

| Date       | Decision | Why |
|------------|----------|-----|
| YYYY-MM-DD | [Exclusion rule, threshold, model choice] | [Reason, and
    ↪ where the evidence is] |
```

```
# Start here

[Two sentences: what this project asks and what the answer was.]

Read in this order:
  1. docs/protocol.md      how the data was collected
  2. notes/decisions.md    every choice, and why
  3. code/01_clean.py      the exclusions live here
  4. notes/known-issues.md what's broken and what I'm unsure about

Then run `bash run_all.sh` and compare
results/tables/key_numbers.csv against results/expected_numbers.csv.
They should match exactly.

The thing to be careful about: [the one trap only you know about].
```

```
# create and activate
python -m venv .venv
source .venv/bin/activate          # Windows: .venv\Scripts\activate

# install
pip install pandas numpy matplotlib scipy

# record
pip freeze > requirements.txt

# restore, elsewhere
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
```

```
conda create -n project-name python=3.11 pandas numpy matplotlib
conda activate project-name

# record what you asked for, not every transitive build string
conda env export --from-history > environment.yml

# restore
conda env create -f environment.yml
conda activate project-name
```

```
# uv
uv venv
uv pip install pandas numpy
uv pip freeze > requirements.txt

# Poetry
poetry init
poetry add pandas numpy
poetry install          # restores from poetry.lock
```

```
install.packages("renv")

renv::init()          # once, in the project directory
                      # creates renv.lock and a private library

# after installing packages as normal
renv::snapshot()      # update renv.lock

renv::restore()       # elsewhere: install exactly what the lock says
renv::status()        # is the library in sync with the lock?
```

```
% code/record_environment.m
fid = fopen('results/logs/environment.txt', 'w');
fprintf(fid, '%s\n', version);
v = ver;
for k = 1:numel(v)
    fprintf(fid, '%s %s\n', v(k).Name, v(k).Version);
end
fclose(fid);
```

```
julia> ]                          # enter package mode
(@v1.11) pkg> activate .          # environment for this project
(project) pkg> add DataFrames Plots
(project) pkg> status
```

```
# restore, elsewhere
julia --project=. -e 'using Pkg; Pkg.instantiate()'
```

```
# Python
import sys, platform, subprocess, datetime
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
out = ROOT / "results" / "logs" / "environment.txt"
out.parent.mkdir(parents=True, exist_ok=True)
with out.open("w") as f:
    f.write(f"date     {datetime.datetime.now().isoformat()}\n")
    f.write(f"python   {sys.version}\n")
    f.write(f"platform {platform.platform()}\n\n")
    f.write(subprocess.run([sys.executable, "-m", "pip", "freeze"],
                           capture_output=True, text=True).stdout)
```

```
# R
writeLines(capture.output(sessionInfo()),
           "results/logs/environment.txt")
```

```
import subprocess
commit = subprocess.run(["git", "rev-parse", "--short", "HEAD"],
                        capture_output=True, text=True).stdout.strip()
dirty = subprocess.run(["git", "status", "--porcelain"],
                       capture_output=True, text=True).stdout.strip()
print(f"code version: {commit}{' (uncommitted changes)' if dirty else ''}")
```

-
-
-
-
-