

PYTHON FROM ZERO SERIES

Python from Zero to Very Advanced

Volume 2 • Intermediate

Chapters 16–43



Badar Mubarok Yogaswara

Contents

Python from Zero to Very Advanced	13
About This Book	15
Who is this book for?	15
What is covered, and what is not	15
What we assume about you	15
How to read what is written in this book	15
Icons in this book	16
The series map: 13 series in three levels	16
Splitting into three volumes	17
How to read: in order or by jumping	18
Beyond the book	18
The Python version used	18
An honest note: what was tested and what was not	18
Ready?	19
Chapter 16 — OOP: Making Moulds for Field Data	21
Class and object	21
Every object has its own data	22
Inheritance: children receive the parent's traits	23
Polymorphism: one command, many ways	24
Encapsulation: guarding data	25
Dunder methods: teaching objects to get along	26
dataclass: data classes with less typing	27
Running Project I1: field data becomes classes	28
Exercises	30
Chapter Summary	30
Chapter 17 — Environments and Dependencies: One Python Room per Project	31
venv: making a new room	31
pip and requirements.txt	33
pyproject.toml: the project's identity card	34
uv: the new-generation package manager	35
Running Project I1: installing inventory	37
Poetry: a rival to uv	38
Anaconda and conda	38

Exercises	40
Chapter Summary	40
Chapter 18 — Testing Basics: Checking Code Automatically with pytest	41
assert: “make sure this is true”	41
pytest and the first test	42
When a test fails	43
Laying out the test folder	43
Fixtures, parametrize, and raises	44
Running some of the tests	46
Running Project I1: tests for the inventory package	47
Exercises	47
Chapter Summary	48
Chapter 19 — Code Quality: Tidy, Logged, and Documented . . .	49
PEP 8: the grammar of Python code	49
ruff: inspector and tidier in one	50
ruff format: tidying the appearance	51
ruff check -fix: automatic repairs	52
pre-commit: an automatic doorkeeper	53
Logging: a program’s diary	55
Logging in the inventory package	56
Debugging: hunting bugs carefully	57
Project documentation: MkDocs and Sphinx	58
Running Project I1: tidying inventory	59
Exercises	60
Chapter Summary	60
Chapter 20 — Git, GitHub, and the Project Workflow: A Time Machine for Your Code	61
Basic git concepts	61
Seeing differences and undoing	62
Branch and merge	63
Conflict: when two people change the same line	64
.gitignore and guarding secrets	65
GitHub: git for working together	67
The daily workflow	67
GitHub Actions: automatic checks on a server	67
Running Project I1: everything together	69
Exercises	70
Chapter Summary	70
Closing Series I1	71

Chapter 21 — Working with APIs: Asking Another Computer for Data	73
HTTP and JSON in five minutes	73
A mock server: our own kitchen	74
The first request with requests	74
JSON and parameters	74
https: the sibling of requests	75
Authentication: showing your “membership card”	75
When things do not go smoothly	76
Status codes and raise_for_status()	76
Timeout: do not wait forever	77
Retry with a pause	77
One call to a real API: Open-Meteo	79
Running Project I2: fetching weather and price data	80
Exercises	80
Chapter Summary	81
Chapter 22 — Regex and Text Processing: Digging Data out of Messy Writing	83
Searching: search, match, findall	83
Ingredients of patterns	84
Groups: taking out the parts	85
Replacing and splitting	86
compile, flags, and VERBOSE	86
Traps: greedy and forgetting None	87
Running Project I2: cleaning field notes	88
Exercises	89
Chapter Summary	89
Chapter 23 — Introduction to numpy and pandas: Smart Tables for Field Data	91
numpy: arrays	91
Whole-array operations	92
pandas: DataFrame	93
Reading and writing files	94
Selecting and filtering	94
Aggregation: summarising per group	95
Joining tables and missing data	96
Running Project I2: merging five sectors, weather, and prices	97
Exercises	99
Chapter Summary	99
Chapter 24 — Databases: A Filing Cabinet That Never Forgets	101
SQLite: a database in one file	101

Never paste user text into SQL	102
Update, delete, and transactions	103
Two tables: JOIN and GROUP BY	104
pandas and SQL	104
SQLAlchemy: one way into many databases	105
PostgreSQL: a server-level database	107
Running Project I2: storing the merged data	108
Exercises	109
Chapter Summary	109
Chapter 25 — Data Science and Analysis: From Tables to Pictures and Guesses	111
Advanced pandas: derived columns and categories	111
Time-ordered data	112
Advanced numpy	113
matplotlib: drawing	114
scikit-learn: your first model	115
Jupyter Notebook and Anaconda	117
Running Project I2: from database to chart	118
Exercises	121
Chapter Summary	121
Closing Series I2	122
Chapter 26 — Statistics with Python: From Numbers to Decisions	123
Descriptive statistics: summarising without deceiving	123
Probability and distributions	124
Sampling and confidence intervals	125
Hypothesis tests: two groups	126
Chi-square: two categories	127
ANOVA: three or more groups	128
Two p-value traps	130
Correlation is not causation	131
Statistical visualisation with seaborn	132
Running Project I3: growth differences between plots and varieties	133
Exercises	134
Chapter Summary	134
Chapter 27 — Regression and Quantitative Modelling: Building Models from Data	137
Simple linear regression: a tree allometric equation	137
Multiple regression and diagnostics	139
Logistic regression: a yes-or-no outcome	140

GLM: count data with Poisson	141
Ridge and Lasso: when predictors are many and alike	142
Non-linear regression: growth curves	143
Mixed-effects models: grouped data	144
Time series and forecasting with ARIMA	145
Spatial regression: GWR	146
Monte Carlo simulation: computing uncertainty	148
Optimisation: finding the best choice	149
System dynamics models and sensitivity analysis	150
Running Project I3: models for our information system	152
Exercises	154
Chapter Summary	154
Chapter 28 – Qualitative and Mixed Methods: Hearing the Stories, Then Counting Them	155
Qualitative, quantitative, or mixed?	155
Our made-up interview data	157
Theme coding	158
Content analysis: word frequency and KWIC	159
TF-IDF: distinctive words	161
Topic modelling: topics from a pile of text	162
Dictionary-based sentiment	163
Embeddings: words become points on a map of meaning	165
Turning qualitative data into quantitative	166
Validity and reliability	167
Running Project I3: interview analysis with mixed methods	168
Exercises	169
Chapter Summary	170
Chapter 29 – Reproducible Research: An Analysis Anyone Can Repeat	171
A tidy Jupyter notebook	172
Quarto: a report built from code	173
A locked environment	174
Snakemake: a workflow that knows what must be repeated	176
DVC: versions for data	178
Sharing data and code	179
Running Project I3: an analysis that can be repeated	181
Exercises	182
Chapter Summary	182
Closing Series I3	182

Chapter 30 — Iterators and Generators: Reading One at a Time, Without Piling Up	183
What for does: the iteration protocol	183
Your own iterator with a class	184
Generators: the short way to make an iterator	185
Generator expressions and proof of saved memory	186
Generators further: <code>itertools</code> , <code>yield from</code> , and a trap	187
Chaining generators into a pipeline	187
Running Project I4: reading a big data file	188
Exercises	190
Chapter Summary	190
 Chapter 31 — Decorators and Closures: Wrapping Functions with Extras	 191
Functions are objects	191
Closures: a function that remembers	192
Your first decorator	193
<code>functools.wraps</code> : keeping the identity	193
A decorator that takes arguments	194
A ready-made decorator and a trap	195
Running Project I4: logging and retrying	196
Exercises	198
Chapter Summary	198
 Chapter 32 — Context Managers: Open, Use, and Surely Close . with on files	 199
Behind the scenes: <code>__enter__</code> and <code>__exit__</code>	200
The short way: <code>@contextmanager</code>	201
<code>__exit__</code> may swallow an error	202
Ready-made tools in <code>contextlib</code>	202
The trap: <code>with</code> on an object that is not a context manager	203
Running Project I4: writing a report safely	203
Exercises	205
Chapter Summary	205
 Chapter 33 — Type Hints: Labelling Data, Checking Before Running	 207
Basic annotations	207
Types for collections, choices, and functions	208
Generics: types that follow the content	209
<code>TypedDict</code> , <code>Literal</code> , and <code>Protocol</code>	210
Running the real <code>mypy</code>	211
<code>Pyright</code> , another checking tool	211

Python 3.14: annotations are evaluated later	212
Running Project I4: typing the project code	213
Exercises	214
Chapter Summary	214
Chapter 34 — Data Validation and Configuration: A Doorkeeper Called Pydantic	215
A dataclass is tidy but does not guard	215
A Pydantic model	216
Clear error messages	217
Extra rules: Field and validators	217
App settings: environment variables and .env	219
JSON, lists of models, and schemas	220
Dataclass or Pydantic?	221
The link to FastAPI and AI agents	221
Running Project I4: validating field data from five sectors	221
Exercises	223
Chapter Summary	223
Chapter 35 — Async and Concurrency: Doing Many Things at Once	225
Two kinds of jobs	225
asyncio : one worker who is good at taking turns	226
gather : many waiting jobs together	226
TaskGroup , time limits, and a concurrency cap	227
Threads: for waiting jobs in ordinary code	228
Processes: for calculating jobs	229
The GIL: why threads do not always speed things up	230
The <i>free-threaded</i> mode	231
Running Project I4: downloading lots of weather data with async	231
Exercises	234
Chapter Summary	235
Closing Series I4	235
Chapter 36 — Web Development: Opening a Data Stall on the Internet	237
How the web works in one paragraph	237
Flask: the smallest stall	238
A REST API with Flask	239
FastAPI: data types that work for you	240
A request body with a Pydantic model	241
Documentation that writes itself	242
A first look at Django	243
A real server, just for a moment	244

Running Project I5: a farm unit data API	245
Exercises	246
Chapter Summary	246
Chapter 37 — Python in the Browser: No Server, No Installing .	247
WebAssembly, in brief	247
Pyodide: Python inside WebAssembly	248
Pyodide straight in an HTML page	249
PyScript: Python inside HTML	249
JupyterLite	250
When it fits, when it does not	250
Running Project I5: a plot area summary shown in the browser	251
Exercises	252
Chapter Summary	252
Chapter 38 — GIS 1: Vector Data: Maps You Can Calculate With	253
Points, lines, polygons: Shapely	253
GeoPandas: a table that has shapes	254
CRS: degrees or metres?	255
Plot areas and pond-to-river distances	256
Buffer and overlay	257
Spatial join: combining by location	258
Installing GIS: pip or conda-forge?	260
Running Project I5: the farm location map	260
Exercises	261
Chapter Summary	261
Chapter 39 — GIS 2: Raster Data: Aerial Photos You Can Calculate With	263
Reading rasters with Rasterio	265
NDVI: measuring greenness	266
Clipping an image with a land boundary	267
Reproject and resample	269
Zonal statistics	269
Running Project I5: NDVI per plot	270
Exercises	271
Chapter Summary	271
Chapter 40 — GIS 3: Maps and Integration: PostGIS and QGIS .	273
Static maps with Matplotlib	273
Interactive maps with folium	274
PostGIS: a database that understands maps	275
Creating tables and loading data	276

Spatial queries	277
PostGIS and GeoPandas	278
PyQGIS: driving QGIS from Python	279
Processing algorithms, styling, and drawing a map	280
QGIS 4 differs from QGIS 3	281
The Python and QGIS workflow	282
Running Project I5: location data in two places	283
Exercises	284
Chapter Summary	284
 Chapter 41 — Automation and Scripting: Let Python Work by Itself	 285
subprocess: calling another program	285
File automation: tidying a folder	287
Infrastructure automation: checking services	288
Scheduling scripts	288
Web scraping: ethics first, code second	290
BeautifulSoup: reading HTML	290
robots.txt in code	292
A polite scraper	292
Scrapy: a framework for big scraping jobs	295
Running Project I5: weekly report automation	297
Exercises	298
Chapter Summary	298
 Chapter 42 — Interfaces: Commands, Windows, and Dashboards	 299
CLI with argparse	299
When the user mistypes	300
Subcommands	302
CLI with Typer	302
Testing a CLI	305
A simple GUI with tkinter	306
An introductory data dashboard	308
Running Project I5: a CLI for business units	309
Exercises	310
Chapter Summary	310
 Chapter 43 — Quick Data Apps and Dashboards: From Script to Someone Else's Screen	 311
Streamlit: a dashboard from one script	312
Gradio: a demo of a function or model	314
Dash: a dashboard you control completely	316
Interactive maps in a dashboard	317
HTMX: lightweight web without a big framework	319

FastAPI: a door for other programs	320
Serving an application to users	322
Running Project I5 (closing): one dataset, many doors	324
Exercises	325
Chapter Summary	325
Closing Series I5	326

Python from Zero to Very Advanced

From complete beginner to expert level, with examples from forest, plantation, rice field, pond, and barn.

Welcome. This book teaches Python from the most basic question (“what is a computer?”) to expert topics (CPython internals, GPUs, distributed systems). It has **68 chapters** in **three levels**, with one project that grows from chapter to chapter: the *Information System for Forestry, Plantations, Farming, Fisheries, and Livestock*.

Volume	Level	Chapters
1	Basic	1–15
2	Intermediate	16–43
3	Advanced	44–68

All code was really run with **Python 3.14.8**, and the output shown is the real result, except for parts that are openly marked as not run. Author: **Badar Mubarak Yogaswara**.

Where to start? Read the next section, [About This Book](#), first: who the book is for, how to read it, and what was tested and what was not. Experienced readers may jump ahead; the path map is in Appendix E.

About This Book

Welcome! This book is called **Python from Zero to Very Advanced**, written by **Badar Mubarak Yogaswara**. “From zero” is no exaggeration: you do not need to have programmed before. “Very advanced” is no exaggeration either: by the end you will touch the inside of Python, GPUs, generative AI, and distributed systems.

Think of this book as a **travel map** from a hut at the edge of a garden to a lookout tower above the forest canopy. You do not have to run. Just take one step at a time, one chapter after another.

Who is this book for?

- **True beginners**: you have never written a single line of code. Chapter 1 starts from the question “what is a computer?”.
- **People who work in the field**: foresters, extension officers, farmers, fish farmers, livestock keepers, students, or researchers who want to process their own data without waiting for someone else.
- **Programmers from other languages** who want to move to Python: you may skip ahead (see “How to read” below).
- **Data, GIS, and AI professionals** who want to tidy up their foundations and see how everything fits together in one project.

What is covered, and what is not

Covered: the basics of the Python language, tidy and tested code, data and the internet, statistics and modelling, web applications, maps (GIS), dashboards, software engineering, big data, machine learning, generative AI and AI agents, up to expert-level Python. All of it is tied together by one **running project**: the *Information System for Forests, Plantations, Agriculture, Fisheries, and Livestock*. The examples take turns drawing from those five sectors.

Not covered: this is not a book about forestry, agriculture, fisheries, or livestock. The field numbers and examples are only **practice material**. Nor is it a complete dictionary of every Python library; we picked the most useful ones and point you to their official documentation.

What we assume about you

We assume only three things: you can use a computer (open files, type, copy), you are curious, and you are willing to **try things yourself**. Reading about programming is like reading about swimming: useful, but you only really feel it after you get into the water.

How to read what is written in this book

A few writing habits stay the same throughout:

- **Code and file names** are written in **monospace** type (equal-width letters), for example `print()` and `data.csv`.
- Every code example is a **script** (a program file) with its **name above the code block**. That name is also the file name in the `code/` folder of your download.
- **Output is always real**. The results printed in the book come from really running that script with **Python 3.14.8**. We never type or edit results by hand. The example below shows it:

```
# About This Book - Checking which Python version runs the examples
import sys
```

```
print("Python version:", sys.version.split()[0])
print("Hello, Land! This is real output, not typed by hand.")
```

```
Python version: 3.14.8
Hello, Land! This is real output, not typed by hand.
```

- Ready-to-copy scripts live in the folder `kode/chNN/` (for example `kode/ch10/`). In the web version, every chapter also has a **zip download** with all of its scripts.
- Technical terms are explained **where they first appear**, printed in **bold**.
- Variable names and text in the examples are in English in this edition so they are easy to read. Python's own error messages are always in English, and the early chapters teach you how to read them.

Icons in this book

Four kinds of boxes appear everywhere. Here is what they really look like:

REMEMBER

Something important to keep in mind. If you only have time to read the boxes, read this kind.

TIP

An easier, faster, or tidier way to do something.

WARNING

A trap that often trips up beginners. We show the real error so you can recognise it.

TECHNICAL STUFF (feel free to skip)

A deeper explanation for the curious. Safe to skip; you can carry on without reading it. Click this box's title to open it.

The series map: 13 series in three levels

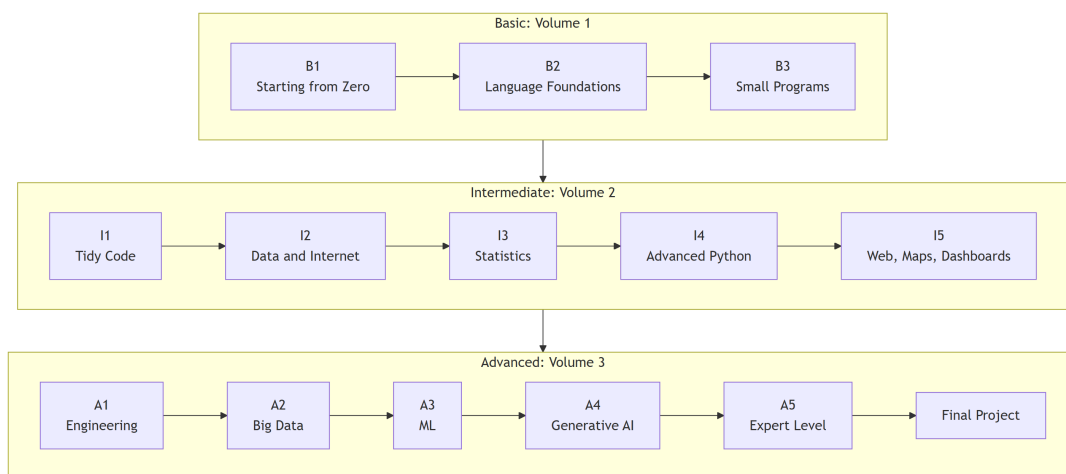
The book has **68 chapters**. The chapters are grouped into **13 series**, and the series into **three levels**. Each series has prerequisites, one running project, and a “bridge” to the next series.

Level	Series	Title	Chapters
Basic	B1	Starting from Zero	1–4
Basic	B2	Foundations of the Python Language	5–10
Basic	B3	Small Programs That Are Useful	11–15
Intermediate	I1	Tidy and Maintainable Code	16–20
Intermediate	I2	Data and the Internet	21–25
Intermediate	I3	Statistics and Modeling	26–29

Level	Series	Title	Chapters
Intermediate	I4	Advanced Python Language	30–35
Intermediate	I5	First Applications: Web, Maps, and Dashboards	36–43
Advanced	A1	Professional Software Engineering	44–49
Advanced	A2	Large-Scale Data and Advanced Geospatial	50–53
Advanced	A3	Machine Learning for Natural Resources	54–59
Advanced	A4	Generative AI and AI Agents	60–63
Advanced	A5	Expert-Level Python	64–68

After Chapter 68 come the **Final Project** (Chapter 69), **Part Ten** (lists of ten), and **Appendices A to F** (installation, cheat sheets, glossary, answer key, learning-path map, further resources).

This book is published in **three volumes**: **Volume 1 (Basic)** holds Chapters 1–15 and Appendix A (installation), **Volume 2 (Intermediate)** holds Chapters 16–43, and **Volume 3 (Advanced)** holds Chapters 44–68, the Final Project, Part Ten, and Appendices A to F. If you hold only one volume, the chapters, project, and appendices missing from it (for example Appendix C, the glossary, and Appendix E, the learning-path map) are in Volume 3.



Order of the series: from B1 to the Final Project.

Splitting into three volumes

A book this thick is easier to hold when split. The author decided (10 October 2026) to divide it into **three volumes** that follow the three levels:

- **Volume 1, Basic:** Chapters 1–15 (series B1 to B3).
- **Volume 2, Intermediate:** Chapters 16–43 (series I1 to I5).
- **Volume 3, Advanced:** Chapters 44–68 (series A1 to A5).

The Final Project, Part Ten, and the Appendices complete the set.

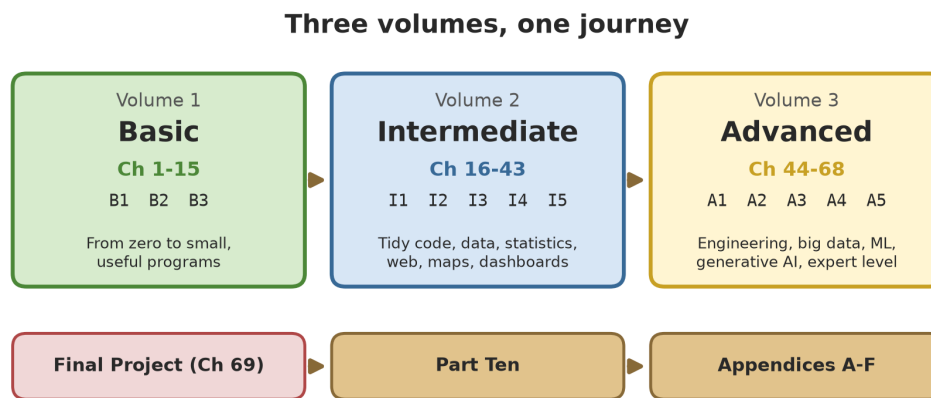


Figure 0.1: Three volumes, one journey.

How to read: in order or by jumping

- **Beginners:** read **in order** from Chapter 1. Each chapter continues from the previous one, and the running project grows bit by bit.
- **Already programming in another language:** start from Series B2 or B3 to get the feel of Python, then jump to the series you need. Check the **prerequisites** at the start of each series.
- **Want a specific topic right away** (say maps or AI): use the table above and **Appendix E (learning-path map)**, which lays out paths for beginners, programmers from other languages, GIS, data/AI, and web.
- **Stuck in a chapter?** Run its example script as it is first, look at the output, and only then change small things. Changing a working example is the fastest way to learn.

Beyond the book

- **Web version:** readable in a browser, with a copy button on every code block.
- **Script downloads:** every chapter has a zip with all of its scripts (the link is at the start of the chapter).
- **Sample data:** small CSV/JSON files are created by the scripts themselves and sit in the `data/` folder of the chapter concerned.
- **Diagrams:** provided as images in the book; the appendices mention other forms when they exist.

The Python version used

This book uses **Python 3.14.8** as its reference (released 30 September 2026; according to the official release schedule PEP 745, which we read, security support for the 3.14 series lasts until about October 2030). **Python 3.15.0** was released only on **9 October 2026**; we checked it on the official python.org download page on 10 October 2026. The 3.15 features are covered as a subsection of **Chapter 47**, and the book's output still comes from 3.14.8.

TIP

You do not have to use 3.14.8. Almost every example runs on Python 3.12 or newer; sections that need a newer feature are marked in their chapter. Check your version with `python --version`. The latest releases are always listed at python.org/downloads, and **version numbers may already have changed** by the time you read this.

An honest note: what was tested and what was not

We want you to know exactly how far this book can be trusted:

- **Tested:** all code was run on **Windows 11** with **Python 3.14.8**, and the output was taken from those runs. Some scripts that do not depend on 3.14-only features were also compared on Python 3.12.
- **Not run:** when some code was **not run** when the book was built (for example because it needs the internet, an account, or hardware that was not available), we say so openly in that chapter together with the reason, and no output is shown.
- **The author's equipment differs from yours.** The parts that use a **GPU**, **QGIS**, **PostgreSQL**, and **Redis** were run on the author's computer (an NVIDIA GPU with 6 GB of VRAM, and isolated local installations). Your computer may differ: library versions, drivers, and the operating system can make results slightly different. If so, read the error message and match it against the relevant chapter.
- **Mock data.** All field data is **fictional data** made by scripts. The numbers for trees, harvests, fish, and livestock are only **practice material**. They are **not technical advice** (not feed rations, planting schedules, or real prices). For decisions in the field, consult the official technical guidance and qualified experts.
- **Links and versions can change.** Web pages, library names, and version numbers were checked when the book was written (October 2026). If a link is dead or a version has moved on, search for the source's name or visit the project's official site, and check its documentation before following an old command.

Ready?

Grab a cup of coffee (or tea), open **Chapter 1**, and begin. Mistakes are normal; an error message is not a punishment but a signpost. Happy planting, code by code!

Chapter 16 — OOP: Making Moulds for Field Data

Series I1 — Tidy and Maintainable Code

In this chapter, you will:

- Make **classes** (moulds) and **objects** (what the mould makes) with their own data and behaviour
- Use **inheritance** so children receive the parent's traits
- Understand **encapsulation** (guarding data) and **polymorphism** (one command, many ways)
- Teach objects to get along with Python through **dunder methods**, then save typing with **dataclass**
- Turn the field data of Series B into a family of classes: the `Unit` parent and its children

All scripts for this chapter are in the `code/ch16/` folder. On the web version, you can also [download them all at once \(zip\)](#).

In Series B, your field data lived in dicts and lists, and the calculations in separate functions. That works, but the bigger the program, the harder a simple question gets: *which function suits which data?* The function `biomass_kg` only makes sense for ponds, yet nothing stops you from handing it cattle data.

Object-oriented programming (OOP) puts **data** and **the functions that suit it** into one package. Think of a cake mould: the **class** is the mould, an **object** is a cake cast from it. One mould makes many cakes, and each cake has its own filling.

Inheritance: the Unit parent, its children, and what each unit holds

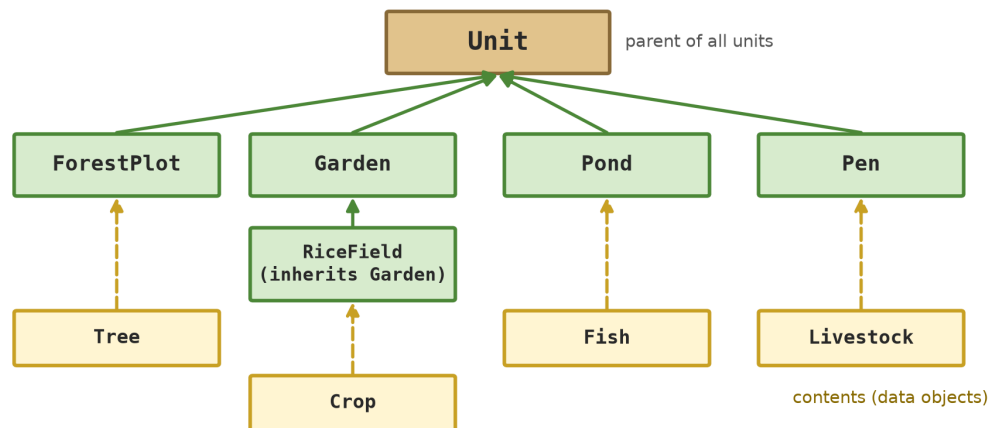


Figure 0.1: The class family of our project: the Unit parent, its children, and the data each unit holds.

Class and object

```
# Chapter 16 - Class and objects: the mould and what it makes
import math
```

```
class Tree:
    """A mould for one tree in a plot."""
```

```

def __init__(self, species, diameter_cm, height_m):
    self.species = species
    self.diameter_cm = diameter_cm
    self.height_m = height_m

def cross_section_cm2(self):
    return math.pi * (self.diameter_cm / 2) ** 2

def category(self):
    if self.diameter_cm >= 40:
        return "large"
    if self.diameter_cm >= 30:
        return "medium"
    return "small"

teak = Tree("Teak", 32.5, 18.0)
meranti = Tree("Meranti", 45.2, 27.5)

print(teak.species, teak.diameter_cm, teak.category())
print(meranti.species, meranti.diameter_cm, meranti.category())
print(f"Teak cross-section: {teak.cross_section_cm2():.1f} cm2")
print(type(teak))

```

```

Teak 32.5 medium
Meranti 45.2 large
Teak cross-section: 829.6 cm2
<class '__main__.Tree'>

```

- `class Tree`: makes the mould. Class names start each word with a capital letter (ForestPlot).
- `__init__` is a special method that runs **automatically when an object is made**. This is where the starting data is set up. (Read “dunder init”: *double underscore*, two underscores on each side.)
- `self` is the object being worked on. `self.species = species` means “keep this value inside this object”.
- A function inside a class is called a **method**. `teak.category()` calls the method for the object `teak`. Python fills in `self` itself.
- `Tree("Teak", 32.5, 18.0)` makes one **object** (also called an *instance*). The cross-section of 829.6 cm² is calculated from the 32.5 cm diameter.

The result of `type(teak)` shows that `teak` really is an object of the class `Tree`.

Every object has its own data

```

# Chapter 16 - Every object has its own data
class Pond:
    unit_kind = "Pond"          # belongs to the class: same for all objects

    def __init__(self, name, fish_count):
        self.name = name        # belongs to the object: differs
        self.fish_count = fish_count

    def stock(self, extra):
        self.fish_count = self.fish_count + extra

pond_a = Pond("A", 800)
pond_b = Pond("B", 500)
pond_a.stock(200)

print(pond_a.name, pond_a.fish_count)

```

```
print(pond_b.name, pond_b.fish_count)
print(pond_a.unit_kind, pond_b.unit_kind)
print(isinstance(pond_a, Pond), isinstance(pond_a, str))
```

```
A 1000
B 500
Pond Pond
True False
```

Pond A starts at 800 and gets 200 more fish, making 1000. Pond B is unchanged at 500. Two objects from the same mould **do not disturb each other**. Note `unit_kind` too: it is written at class level, so all objects **share** it.

⚠ WARNING

The most common trap when first writing methods: forgetting `self`.

```
# Chapter 16 - Trap: forgetting self
class Tree:
    def __init__(self, species):
        self.species = species

    def greet():
        # forgot self!
        print("Hello, I am a tree")

teak = Tree("Teak")
teak.greet()
```

```
Traceback (most recent call last):
  File "03_forgot_self.py", line 11, in <module>
    teak.greet()
    ~~~~~^
TypeError: Tree.greet() takes 0 positional arguments but 1 was given
```

The message is confusing: “takes 0 positional arguments but 1 was given”. The argument meant is the object itself, which Python sends automatically. Without `self` in the parameter list, there is nowhere to receive it.

Inheritance: children receive the parent’s traits

A forest plot, garden, pond, and pen all have a name and an area. Writing that again in every class is wasteful. The solution is **inheritance**: make a **parent** class holding what they share, then **child** classes that inherit from it.

```
# Chapter 16 - Inheritance: children inherit from the parent
class Unit:
    """Parent: every unit has a name and an area."""

    def __init__(self, name, area_ha):
        self.name = name
        self.area_ha = area_ha

    def info(self):
        return f"{self.name} ({self.area_ha} ha)"

class ForestPlot(Unit):
    def __init__(self, name, area_ha, stand):
```

```

        super().__init__(name, area_ha)    # parent's part first
        self.stand = stand                 # then the child's own part

class Garden(Unit):
    pass                                # adds nothing

plot = ForestPlot("Plot A", 2.5, "Teak")
garden = Garden("Coffee Garden", 1.2)
print(plot.info(), "| stand:", plot.stand)
print(garden.info())
print(isinstance(plot, Unit),      isinstance(Garden, Unit),      isinstance(garden,
ForestPlot))

```

```

Plot A (2.5 ha) | stand: Teak
Coffee Garden (1.2 ha)
True True False

```

- `class ForestPlot(Unit):` means “ForestPlot is a child of Unit”.
- `super().__init__(...)` calls the parent’s `__init__`, so the parent’s part gets done too. The child then adds its own part (`stand`).
- `Garden` adds nothing, yet still has `name`, `area_ha`, and `info()` from `Unit`.
- `isinstance(plot, Unit)` is `True`: a forest plot *is* a unit. The reverse does not hold: a garden is not a forest plot (`False`).

Polymorphism: one command, many ways

The word sounds scary, but the idea is simple: **children may carry out the same command in different ways**. You just say `summary()`, and each unit knows how to work it out.

```

# Chapter 16 - Polymorphism: one command, many ways
class Unit:
    def __init__(self, name):
        self.name = name

    def output(self):
        raise NotImplementedError("the child must fill in output()")

    def summary(self):
        return f"{self.name}: {self.output()}"

class Pond(Unit):
    def __init__(self, name, count, weight_g):
        super().__init__(name)
        self.count, self.weight_g = count, weight_g

    def output(self):
        return f"biomass {self.count * self.weight_g / 1000:.1f} kg"

class Pen(Unit):
    def __init__(self, name, cattle_weights_kg):
        super().__init__(name)
        self.cattle_weights_kg = cattle_weights_kg

    def output(self):
        return f"total weight {sum(self.cattle_weights_kg):.1f} kg"

```

```

units = [Pond("Pond A", 800, 540.5), Pen("Pen 1", [350, 412.5, 389.0])]
for unit in units:
    print(unit.summary())          # same command, the result adapts

try:
    Unit("Plain").summary()
except NotImplementedError as e:
    print("The parent alone is not enough:", e)

```

```

Pond A: biomass 432.4 kg
Pen 1: total weight 1151.5 kg
The parent alone is not enough: the child must fill in output()

```

The `for` loop calls `unit.summary()` without caring whether it holds a pond or a pen. The pond sum ($800 \times 540.5 \text{ g} = 432.4 \text{ kg}$) and the pen sum ($350 + 412.5 + 389.0 = 1151.5 \text{ kg}$) differ, but the caller is the same. The parent `Unit` hands `output()` over to the children with `raise NotImplementedError`: “the child must fill this in”. That is why `Unit("Plain")` fails with a clear message.

Encapsulation: guarding data

Encapsulation means data is guarded **inside** the object, and outside access goes through a door with a guard. In Python, a leading underscore (`_water_ph`) is an **agreement**: “this is inside business, do not touch it from outside”. The door guard is a **property**.

```

# Chapter 16 - Encapsulation: data is guarded inside the object
class Pond:
    def __init__(self, name, water_ph):
        self.name = name
        self._water_ph = None      # leading _ means: do not touch from outside
        self.water_ph = water_ph  # via the property, so it gets checked

    @property
    def water_ph(self):
        return self._water_ph

    @water_ph.setter
    def water_ph(self, value):
        if not 0 <= value <= 14:
            raise ValueError(f"pH must be 0 to 14, not {value}")
        self._water_ph = value

pond = Pond("A", 7.2)
print("Initial pH:", pond.water_ph)
pond.water_ph = 6.8
print("New pH:", pond.water_ph)

try:
    pond.water_ph = 17
except ValueError as e:
    print("Rejected:", e)
print("pH unchanged:", pond.water_ph)

```

```

Initial pH: 7.2
New pH: 6.8
Rejected: pH must be 0 to 14, not 17
pH unchanged: 6.8

```

From outside, `pond.water_ph` looks like an ordinary variable. But each time the value changes, Python runs the check in the `setter`. A pH of 17 is rejected with a `ValueError`, and the old value (6.8) stays safe.

i REMEMBER

Python does not truly lock data. The `_` prefix is only an agreement between programmers. The strength of `property` is the **check**: wrong data cannot get in through the official door.

Dunder methods: teaching objects to get along

Methods with double underscores like `__init__` are called **dunder methods**. Python calls them automatically at certain moments. By filling them in, your objects work with `print`, `==`, `sorted`, and `len`.

```
# Chapter 16 - Dunder methods: teaching objects to get along with Python
class Tree:
    def __init__(self, species, diameter_cm):
        self.species = species
        self.diameter_cm = diameter_cm

    def __repr__(self):
        # the technical display
        return f"Tree({self.species!r}, {self.diameter_cm})"

    def __str__(self):
        # the display for humans
        return f"{self.species} {self.diameter_cm} cm"

    def __eq__(self, other):
        return (self.species, self.diameter_cm) == (other.species, other.diameter_cm)

    def __lt__(self, other):
        # so it can be sorted
        return self.diameter_cm < other.diameter_cm

class Plot:
    def __init__(self, trees):
        self.trees = list(trees)

    def __len__(self):
        return len(self.trees)

a, b, c = Tree("Teak", 32.5), Tree("Meranti", 45.2), Tree("Mahogany", 28.0)
print(a)
print(repr(a))
print(a == Tree("Teak", 32.5), a == b)
print(sorted([a, b, c]))
print(len(Plot([a, b, c])), "trees")
```

```
Teak 32.5 cm
Tree('Teak', 32.5)
True False
[Tree('Mahogany', 28.0), Tree('Teak', 32.5), Tree('Meranti', 45.2)]
3 trees
```

Method	Called when	Example
<code>__str__</code>	<code>print(object)</code>	display for readers
<code>__repr__</code>	<code>repr(object)</code> , display inside a list	display for programmers

Method	Called when	Example
<code>__eq__</code>	<code>a == b</code>	compares contents, not addresses
<code>__lt__</code>	<code>a < b, sorted(...)</code>	sorts by diameter
<code>__len__</code>	<code>len(object)</code>	number of items inside

The `sorted` result runs from diameter 28.0 to 32.5 to 45.2 because `__lt__` compares diameters.

dataclass: data classes with less typing

A class that only holds data (like `Tree`) forces us to write `__init__`, `__repr__`, and `__eq__` again and again. The built-in `dataclasses` module makes them automatically from a **list of typed columns**.

```
# Chapter 16 - dataclass: data classes with less typing
from dataclasses import dataclass, field

@dataclass(order=True)
class Tree:
    species: str
    diameter_cm: float
    healthy: bool = True

@dataclass
class Plot:
    name: str
    trees: list = field(default_factory=list)    # a fresh list for each plot

a = Tree("Teak", 32.5)
b = Tree("Mahogany", 28.0, healthy=False)
print(a)                                     # __repr__ made automatically
print(a == Tree("Teak", 32.5))              # __eq__ made automatically
print(b < a)                                # order=True: compared column by column ("Mahogany"
comes before "Teak")

plot_a, plot_b = Plot("A"), Plot("B")
plot_a.trees.append(a)
print(len(plot_a.trees), len(plot_b.trees))
```

```
Tree(species='Teak', diameter_cm=32.5, healthy=True)
True
True
1 0
```

- Write columns with a **type hint** (`species: str`). Defaults are set with `(healthy: bool = True)`.
- `@dataclass` makes `__init__`, `__repr__`, and `__eq__`. The option `order=True` adds ordering comparisons. “Mahogany” sorts before “Teak” because the first column (`species`) is compared first, alphabetically.
- For a list default, use `field(default_factory=list)` so each object gets a **new** list. The final `1 0` proves plots A and B do not share one list.

TIP

Industry habit is to name classes, functions, and variables in English. In this book the Indonesian edition uses Indonesian names for easy reading, but in a real team, follow your team’s convention.

Running Project I1: field data becomes classes

Now all the ingredients work together. Four `dataclasses` hold data (`Tree`, `Crop`, `Fish`, `Livestock`), the `Unit` parent holds what all units share, and five children (`ForestPlot`, `Garden`, `RiceField`, `Pond`, `Pen`) fill in `output()` their own way. `RiceField` is a child of `Garden`: a rice field is a garden that is also counted per hectare.

```
# Chapter 16 - Running project I1: field data becomes classes
from dataclasses import dataclass

@dataclass
class Tree:
    species: str
    diameter_cm: float
    height_m: float
    healthy: bool = True

@dataclass
class Crop:
    commodity: str
    block: str
    age_days: int
    yield_kg: float

@dataclass
class Fish:
    species: str
    count: int
    avg_weight_g: float
    water_ph: float

    def biomass_kg(self):
        return self.count * self.avg_weight_g / 1000

@dataclass
class Livestock:
    species: str
    age_months: int
    weight_kg: float

class Unit:
    """Parent of all land or enterprise units."""

    kind = "Unit"
    measure_unit = ""

    def __init__(self, name, area_ha):
        if area_ha <= 0:
            raise ValueError("area_ha must be greater than 0")
        self.name = name
        self.area_ha = area_ha
        self._items = []

    def add(self, item):
        self._items.append(item)
```

```

@property
def items(self):
    return tuple(self._items)

def output(self):
    raise NotImplementedError

def summary(self):
    return (f"{self.kind} {self.name}: {len(self._items)} record(s), "
            f"{self.output():.1f} {self.measure_unit}")

def __len__(self):
    return len(self._items)

class ForestPlot(Unit):
    kind, measure_unit = "Forest plot", "healthy trees"

    def output(self):
        return sum(1 for t in self._items if t.healthy)

class Garden(Unit):
    kind, measure_unit = "Garden", "kg"

    def output(self):
        return sum(c.yield_kg for c in self._items)

class RiceField(Garden):
    # a rice field = a garden also counted
    per hectare
    kind = "Rice field"

    def productivity(self):
        return self.output() / self.area_ha

    def summary(self):
        return super().summary() + f" ({self.productivity():.0f} kg/ha)"

class Pond(Unit):
    kind, measure_unit = "Pond", "kg biomass"

    def output(self):
        return sum(f.biomass_kg() for f in self._items)

class Pen(Unit):
    kind, measure_unit = "Pen", "kg weight"

    def output(self):
        return sum(a.weight_kg for a in self._items)

plot = ForestPlot("A", 2.5)
for t in [Tree("Teak", 32.5, 18.0), Tree("Meranti", 45.2, 27.5),
          Tree("Mahogany", 28.0, 15.0, healthy=False)]:
    plot.add(t)

garden = Garden("Coffee", 1.2)

```

```

for block, kg in [("A", 245.5), ("B", 310.0)]:
    garden.add(Crop("coffee", block, 400, kg))

rice_field = RiceField("Lowland", 0.5)
rice_field.add(Crop("rice", "1", 110, 1250.0))

pond = Pond("A", 0.04)
pond.add(Fish("catfish", 800, 540.5, 7.2))

pen = Pen("1", 0.1)
for species, months, kg in [("cattle", 30, 350), ("cattle", 36, 412.5), ("cattle",
33, 389.0)]:
    pen.add(Livestock(species, months, kg))

for unit in [plot, garden, rice_field, pond, pen]:
    print(unit.summary())

```

```

Forest plot A: 3 record(s), 2.0 healthy trees
Garden Coffee: 2 record(s), 555.5 kg
Rice field Lowland: 1 record(s), 1250.0 kg (2500 kg/ha)
Pond A: 1 record(s), 432.4 kg biomass
Pen 1: 3 record(s), 1151.5 kg weight

```

Trace the numbers: plot A has 3 trees, 1 unhealthy, so 2 healthy trees. The coffee garden adds $245.5 + 310.0 = 555.5$ kg. The rice field's 1250 kg on 0.5 ha is 2500 kg/ha. The pond's 432.4 kg and the pen's 1151.5 kg match the Series B calculations. All figures are fictional.

Note that the list `[plot, garden, rice_field, pond, pen]` holds five kinds of objects, yet one loop and one `summary()` command handle them all. Adding a new unit type (say `Hatchery`) takes just one new child. The next chapters install this project in its own environment, test it, tidy it, and keep it in git.

Exercises

1. Add a method `full_info()` to the class `Tree` in `01_tree_class.py` that returns text like "Teak, 32.5 cm, category medium".
2. Make your own class `Garden` (not a child) with name and a list `harvest_kg`, then fill in `__len__` and `__str__`. Print one object and call `len()` on it.
3. In `04_inheritance.py`, add a class `Pond(Unit)` with an attribute `water_ph`. Make one object and check `isinstance(pond, Unit)`.
4. **Challenge:** add a class `Pen` to `05_polymorphism.py` that fills in `output()` with the *average* weight rather than the total. Also add a new class `Hatchery(Unit)`; make sure the old loop needs no change.

Chapter Summary

- **Class** = mould, **object** = what it casts. `__init__` sets the starting data; **self** is the object being worked on.
- **Inheritance:** `class Child(Parent); super().__init__(...)` calls the parent.
- **Polymorphism:** same command, the result adapts to the kind of object.
- **Encapsulation:** the `_` prefix as an agreement, `property` as a guard that checks.
- **Dunder methods** (`__str__`, `__repr__`, `__eq__`, `__lt__`, `__len__`) make objects comfortable with Python.
- **@dataclass** generates `__init__`, `__repr__`, and `__eq__` from a list of columns.

Next, in Chapter 17: environments and dependencies, how to give each project its own Python "room" and manage the packages it needs.

Chapter 17 — Environments and Dependencies: One Python Room per Project

Series I1 — Tidy and Maintainable Code

In this chapter, you will:

- Understand why each project needs its own **environment**
- Make and use a **venv**, install packages with **pip**, and keep the list in `requirements.txt`
- Read and write `pyproject.toml`, the project's "identity card"
- Meet **uv** (the new-generation project manager), **poetry**, and **Anaconda/conda**, including the Anaconda licence note
- Install the `inventory` package of the running project in its own environment

All scripts for this chapter are in the `code/ch17/` folder. On the web version, you can also [download them all at once \(zip\)](#).

In Chapter 16, the `inventory` project was born as a set of classes. From Chapter 18 on, it will use outside libraries (pytest, ruff). Here the classic problem appears. Project A needs an old `pandas`, project B needs a new one. Install both into one Python and one of them breaks. It is like two farmers sharing one fertiliser store: the moment one changes the contents, the other is disturbed.

The solution is an **environment**: a separate Python "room" for each project, with its own packages. A **dependency** is an outside package your project needs.

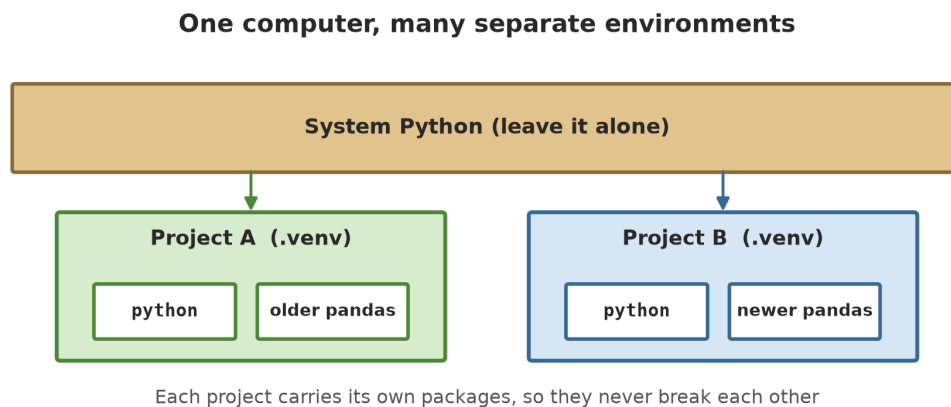


Figure 0.1: One computer, many environments: each project carries its own packages.

venv: making a new room

venv is Python's built-in module for making environments. The terminal command: `python -m venv .venv` (the name `.venv` is only a habit). The script below makes one through code and inspects it.

```
# Chapter 17 - Making a venv: a Python "room" just for one project
import os
import shutil
import subprocess
```

```

import sys
import venv
from pathlib import Path

work = Path(__file__).parent / "_tmp"
shutil.rmtree(work, ignore_errors=True)
work.mkdir()
location = work / "sample_venv"

venv.create(location, with_pip=True)          # same as: python -m venv sample_venv

bin_folder = location / ("Scripts" if os.name == "nt" else "bin")
print("Contents of the venv folder:", sorted(p.name for p in location.iterdir()))
print("Important files present:", [
    name for name in ("python", "pip", "activate")
    if any(p.stem == name for p in bin_folder.iterdir())
])

cfg = (location / "pyenv.cfg").read_text(encoding="utf-8").splitlines()
print([line for line in cfg if line.startswith("include-system-site-packages")])

venv_python = bin_folder / ("python.exe" if os.name == "nt" else "python")
code = "import sys; print(sys.prefix != sys.base_prefix)"
inside = subprocess.run([venv_python, "-c", code], capture_output=True,
text=True).stdout.strip()
system_python = Path(sys.base_prefix) / ("python.exe" if os.name == "nt" else "bin/
python3")
outside = subprocess.run([system_python, "-c", code], capture_output=True,
text=True).stdout.strip()
print("Python inside the venv is separate from the system?", inside)
print("System Python (parent of the venv) is separate?", outside)

```

```

Contents of the venv folder: ['Include', 'Lib', 'Scripts', 'pyenv.cfg']
Important files present: ['python', 'pip', 'activate']
['include-system-site-packages = false']
Python inside the venv is separate from the system? True
System Python (parent of the venv) is separate? False

```

- The new folder holds `Scripts` (named `bin` on Linux/macOS), `Lib` (where packages are kept), `Include`, and `pyenv.cfg`. Inside `Scripts` sit `python`, `pip`, and the `activate` scripts.
- `include-system-site-packages = false` means packages in the system Python are **not visible**. This room is truly clean.
- The Python inside the venv reports `sys.prefix != sys.base_prefix = True`: it knows it lives in a venv. The system Python reports `False`.

To use it in a terminal, **activate** it first. Activation puts the `Scripts` folder at the front of the command search path, so the `python` and `pip` that run are the venv's. According to the official Python documentation (read 2026-10-10):

Terminal	Activation command
Windows, cmd	<code>.venv\Scripts\activate.bat</code>
Windows, PowerShell	<code>.venv\Scripts\Activate.ps1</code>
Linux/macOS, bash/zsh	<code>source .venv/bin/activate</code>

To leave, type `deactivate`. Activation is **optional**: you can also call `.venv\Scripts\python.exe` directly, which is what the scripts in this chapter do.

⚠ WARNING

Do not **move** or **copy** a venv folder elsewhere, and do not keep it in git. The Python documentation explains that scripts inside hold absolute paths, so a moved venv may break. The right way: throw it away and **rebuild** it from the package list (`requirements.txt` or `pyproject.toml`). Python 3.13 and later even put a `.gitignore` inside the venv automatically.

pip and requirements.txt

pip is Python's built-in package installer; packages are downloaded from **PyPI** (the *Python Package Index*, the official package store). The script below proves venv isolation: the package `requests` is installed in one venv, its list is frozen (`pip freeze`) into **requirements.txt**, and that file builds a second venv.

```
# Chapter 17 - pip inside a venv: install, freeze, reinstall from requirements.txt
import os
import shutil
import subprocess
import venv
from pathlib import Path

work = Path(__file__).parent / "_tmp"
shutil.rmtree(work, ignore_errors=True)
work.mkdir()

def python_of(location):
    return location / ("Scripts/python.exe" if os.name == "nt" else "bin/python")

def run(location, *arguments):
    return subprocess.run([python_of(location), *arguments], capture_output=True,
text=True)

venv_a = work / "venv_a"
venv.create(venv_a, with_pip=True)

result = run(venv_a, "-c", "import requests")
print("Before installing:", result.stderr.strip().splitlines()[-1])

run(venv_a, "-m", "pip", "install", "--quiet", "--disable-pip-version-check",
"requests")
result = run(venv_a, "-c", "import requests; print('imported successfully')")
print("After installing:", result.stdout.strip())

frozen = run(venv_a, "-m", "pip", "freeze").stdout
(work / "requirements.txt").write_text(frozen, encoding="utf-8")
print("--- requirements.txt ---")
print(frozen.strip())

venv_b = work / "venv_b"
venv.create(venv_b, with_pip=True)
run(venv_b, "-m", "pip", "install", "--quiet", "--disable-pip-version-check",
"-r", str(work / "requirements.txt"))
frozen_b = run(venv_b, "-m", "pip", "freeze").stdout
print("venv_b identical to venv_a?", frozen_b == frozen)
```

```

Before installing: ModuleNotFoundError: No module named 'requests'
After installing: imported successfully
--- requirements.txt ---
certifi==2026.7.22
charset-normalizer==3.5.2
idna==3.20
requests==2.34.2
urllib3==2.8.0
venv_b identical to venv_a? True

```

This code was **really run once** by the author and the output saved, because it needs the internet and contains version numbers that keep changing. So the version numbers above are the state at the time of writing; on your computer they will differ.

Trace the result:

- **Before installing**, `import requests` fails with `ModuleNotFoundError`. A new room is indeed empty.
- **After** `pip install requests`, the import works.
- `pip freeze` records **five** packages, not one: `requests` brought four other dependencies (`certifi`, `charset-normalizer`, `idna`, `urllib3`). Each is pinned to an exact version (`==`).
- The second venv, built from `requirements.txt`, is identical to the first (`True`). That is the point of this file: a colleague or a server can build the same room.

TIP

Install packages with `python -m pip install ...`, not just `pip install ...`. The first form makes sure the pip used belongs to the Python you are calling. That avoids the “installed into the wrong Python” mistake.

pyproject.toml: the project’s identity card

`requirements.txt` is only a package list. Modern projects keep **everything** in one file, **pyproject.toml**: name, version, required Python, dependencies, and tool settings (`pytest`, `ruff`). The format is **TOML**: sections are written `[name]`, contents `key = value`. This is the file of the `inventory` package:

```

[build-system]
requires = ["setuptools>=61"]
build-backend = "setuptools.build_meta"

[project]
name = "inventory"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = []

[project.optional-dependencies]
dev = ["pytest", "ruff"]

[tool.ruff]
line-length = 88

```

- `[build-system]` tells the installer how to “build” the package (here with `setuptools`).
- `[project]` holds the project identity. `dependencies` are the required packages (empty: `inventory` only uses the standard library). `optional-dependencies` holds extra groups, here `dev` for developer tools.
- `[tool.xxx]` is a settings space for tools, such as `[tool.ruff]` and `[tool.pytest.ini_options]` (Chapter 18).

Python 3.11 and later has the built-in module `tomllib` to read it:

```
# Chapter 17 - Reading pyproject.toml with tomllib (built into Python 3.11+)
import tomllib
from pathlib import Path

file = Path(__file__).parent / "project" / "pyproject.toml"
with open(file, "rb") as f:          # tomllib requires binary mode
    data = tomllib.load(f)

project = data["project"]
print("Project name      :", project["name"])
print("Version           :", project["version"])
print("Needs Python      :", project["requires-python"])
print("Dependencies      :", project["dependencies"])
print("Dev dependencies   :", project["optional-dependencies"]["dev"])
print("Build tool         :", data["build-system"]["build-backend"])
print("[tool.*] sections:", sorted(data["tool"]))
print("Ruff line length  :", data["tool"]["ruff"]["line-length"])
```

```
Project name      : inventory
Version           : 0.1.0
Needs Python      : >=3.12
Dependencies      : []
Dev dependencies  : ['pytest', 'ruff']
Build tool        : setuptools.build_meta
[tool.*] sections: ['ruff', 'setuptools']
Ruff line length  : 88
```

Remember: TOML is read as an ordinary dict. `data["project"]["name"]` is a dict inside a dict (Chapter 9).

uv: the new-generation package manager

uv is a Python package and project manager made by Astral (the maker of ruff, which we use in Chapter 19). It does the work of `venv` + `pip` + version locking in one command, and is very fast. Based on the official uv documentation (read 2026-10-10) and tries on the author's computer:

Command	Function
<code>uv init</code>	makes a new project (<code>pyproject.toml</code> , <code>.python-version</code> , <code>main.py</code>)
<code>uv add package</code>	adds a dependency to <code>pyproject.toml</code> , makes <code>.venv</code> and <code>uv.lock</code>
<code>uv run command</code>	runs a command in the project environment, making sure it is up to date
<code>uv lock / uv sync</code>	updates the lock file / aligns the environment with the lock
<code>uv tree</code>	shows the dependency tree
<code>uv venv, uv pip ...</code>	compatible replacements for <code>python -m venv</code> and <code>pip</code>

```
# Chapter 17 - uv: a new project from scratch (init, add, run, tree)
import shutil
import subprocess
from pathlib import Path

work = Path(__file__).parent / "_tmp" / "uv_demo"
shutil.rmtree(work, ignore_errors=True)
work.mkdir(parents=True)
```



```

def uv(*arguments):
    result = subprocess.run(["uv", *arguments], cwd=work, capture_output=True,
text=True)
    return result.stdout.strip()

uv("init", "--name", "inventory-demo", "--no-readme", "--vcs", "none", "--python",
"3.12")
print("Files after uv init:", sorted(p.name for p in work.iterdir()))

uv("add", "requests")
print("Files after uv add :", sorted(p.name for p in work.iterdir()))

lines = (work / "pyproject.toml").read_text(encoding="utf-8").splitlines()
print("--- pyproject.toml (dependencies) ---")
print("\n".join(lines[lines.index("dependencies = [") : lines.index(")]")+1]))

print("--- uv run ---")
print(uv("run", "python", "-c", "import requests; print('requests is ready to use')"))

print("--- uv tree ---")
print(uv("tree"))

```

```

Files after uv init: ['.python-version', 'main.py', 'pyproject.toml']
Files after uv add : ['.python-version', '.venv', 'main.py', 'pyproject.toml',
'uv.lock']
--- pyproject.toml (dependencies) ---
dependencies = [
    "requests>=2.34.2",
]
--- uv run ---
requests is ready to use
--- uv tree ---
inventory-demo v0.1.0
├─ requests v2.34.2
│   ├── certifi v2026.7.22
│   ├── charset-normalizer v3.5.2
│   ├── idna v3.20
│   └─ urllib3 v2.8.0

```

This too was really run and the output saved (it needs the internet and contains version numbers). Notice:

- `uv init` makes only three files. `.venv` and `uv.lock` are born only when `uv add` runs. `uv.lock` is a **lock file** holding the exact versions of all packages, managed by uv and not to be edited by hand (uv documentation).
- `uv add requests` writes `"requests>=2.34.2"` into `pyproject.toml`: a lower bound, not a fixed price. The exact version lives in `uv.lock`.
- `uv tree` shows the tree: `requests` on top, four dependencies beneath it, the same as the `pip freeze` result earlier.

i REMEMBER

Two kinds of list have different jobs. **pyproject.toml** says “which packages I need” (loose: `>=2.34.2`). A **lock file** (`uv.lock`, a frozen `requirements.txt`) says “the exact versions installed” so everyone gets the same result.

Running Project I1: installing inventory

Now the `inventory` package from Chapter 16 is installed in its own environment with `uv`. The `-e` option (*editable*) makes Python read code directly from the project folder, so code changes show up at once without reinstalling.

```
# Chapter 17 - Running project I1: installing the inventory package in its own
environment
import os
import shutil
import subprocess
from pathlib import Path

parent = Path(__file__).parent
work = parent / "_tmp" / "install"
shutil.rmtree(work, ignore_errors=True)
work.mkdir(parents=True)
copy = work / "project"
shutil.copytree(parent / "project", copy)

environment = work / "venv_inventory"
subprocess.run(["uv", "venv", str(environment), "--quiet"], check=True)
python = environment / ("Scripts/python.exe" if os.name == "nt" else "bin/python")

# -e = "editable": code in the project folder is read directly, no reinstall needed
subprocess.run(["uv", "pip", "install", "--quiet", "--python", str(python), "-e",
str(copy)],
               check=True)

code = """
from importlib.metadata import version
from pathlib import Path

import inventory
from inventory.sample import make_sample

print("Installed: inventory", version("inventory"))
print("Code read from the project folder?",
Path(inventory.__file__).parent.parent.name == "project")
for unit in make_sample():
    print(unit.summary())
"""
result = subprocess.run([python, "-c", code], capture_output=True, text=True)
print(result.stdout.strip())
```

```
Installed: inventory 0.1.0
Code read from the project folder? True
Forest plot A: 3 record(s), 2.0 healthy trees
Garden Coffee: 2 record(s), 555.5 kg
Rice field Lowland: 1 record(s), 1250.0 kg (2500 kg/ha)
Pond A: 1 record(s), 432.4 kg biomass
Pen 1: 3 record(s), 1151.5 kg weight
```

The storyline: a copy of the project is placed in `_tmp/`, `uv venv` makes an empty room, `uv pip install -e` installs the package (`uv` downloads the `setuptools` build tool, which is why it needs the internet, and the output is saved from one run). Then **inside that room** `inventory` can be imported from anywhere, and the result matches Chapter 16 exactly: forest plot A, coffee garden, lowland rice field, pond, and pen.

Poetry: a rival to uv

poetry is an older, popular Python project manager; it also uses `pyproject.toml` and makes a lock file, `poetry.lock`. From the official poetry documentation (read 2026-10-10): `poetry new` makes a project folder, `poetry add` adds a dependency and installs it, `poetry install` installs all dependencies, and `poetry run` runs a command in the project environment. Dependencies are written in `project.dependencies` (the PEP 621 standard); older poetry used `tool.poetry.dependencies`.

```
# Chapter 17 - The basic poetry flow (an example only; not run when the book is built)
import subprocess

commands = [
    ["poetry", "new", "inventory-poetry"],    # makes a new project folder
    ["poetry", "add", "requests"],            # adds a dependency + locks versions
    (poetry.lock)
    ["poetry", "install"],                    # installs all project dependencies
    ["poetry", "run", "pytest"],              # runs a command in the project environment
]
for c in commands:
    print("$", " ".join(c))
    subprocess.run(c, check=True)
```

This code was **not run** when the book was built because poetry is not installed on the author's computer (see the result of `06_check_tools.py` below), so no output is shown. The commands were matched against the official documentation but not tested directly.

Anaconda and conda

Anaconda is another way to install and manage Python, popular in data science. Three terms are often mixed up:

- **conda**: a command-line tool to make environments and install packages. Its documentation says conda provides “package, dependency, and environment management for any language”, so not only Python packages.
- **Anaconda Distribution**: an installer bundle that brings conda, Python, **Anaconda Navigator** (a windowed app to make environments and open tools without typing commands), and 600+ ready-made data science packages, about 9.7 GB.
- **Miniconda**: the slim version: conda, Python, 130+ packages, no Navigator, about 900 MB.

(Size figures from the official Anaconda pages, read 2026-10-10.)

How it differs from python.org: the python.org installer gives only Python; the rest you handle with `venv` and `pip`. Anaconda gives Python plus an environment manager and many packages at once. Its packages come from conda **channels**, not from PyPI.

The flow in a terminal (from the official conda documentation, read 2026-10-10):

```
conda create --name inventory python numpy    # make an environment holding Python
and numpy
conda activate inventory                      # enter the environment
conda install --name inventory matplotlib    # install another package
conda info --envs                            # list environments
conda deactivate                             # leave
```

```
# Note: "conda activate inventory" is typed directly in the terminal, not through
a script.
```

```
# Chapter 17 - The basic conda flow (an example only; not run when the book is built)
import subprocess
```

```
commands = [
    ["conda", "create", "--name", "inventory", "python", "numpy"],    # make an
environment + contents
    ["conda", "install", "--name", "inventory", "numpy"],              # install a package
```

```

["conda", "info", "--envs"],
]
for c in commands:
    print("$", " ".join(c))
    subprocess.run(c, check=True)

```

This code was **not run** when the book was built because conda is not installed on the author's computer, so no output is shown and **the whole conda part of this chapter has not been tested directly**. (conda activate is deliberately absent from the script: it is typed in the terminal.)

```

# Chapter 17 - Checking which tools exist on the computer
import shutil

for tool in ["python", "pip", "uv", "git", "poetry", "conda"]:
    path = shutil.which(tool)
    print(f"{tool:7}", "present" if path else "NOT present")

```

```

python  present
pip     present
uv      present
git     present
poetry  NOT present
conda   NOT present

```

This output was taken once on the author's computer and differs if run on yours. Here `uv` and `git` are present, `poetry` and `conda` are not.

conda vs pip: pip installs from PyPI, conda from conda channels. The conda documentation advises: if you need both in one environment, **install as much as possible with conda first, then pip** ("Use pip only after conda"). **conda-forge** is a community-run channel, separate from the `defaults` channel of Anaconda, Inc. The conda-forge documentation suggests Miniforge as the easiest way to start.

Jupyter: the official Anaconda pages say Jupyter Notebook can be launched from Navigator. (They do not spell out whether Jupyter is bundled; check after installing.)

⚠ WARNING

Anaconda licence. Anaconda is not free for everyone. According to the Anaconda Terms of Service (<https://www.anaconda.com/legal/terms/terms-of-service>, the stated date is 15 July 2025, read 2026-10-10), use of the platform is **free** if you are: (1) an individual for personal, non-commercial purposes, (2) acting for or with an eligible academic institution, (3) acting for or with an eligible non-profit and research organisation, or (4) acting for a **for-profit organisation with 200 or fewer** employees or contractors (including affiliates). A for-profit organisation with **more than 200** employees or contractors that does not fall in a free category must subscribe to the Business plan. The pricing page (<https://www.anaconda.com/pricing>, read 2026-10-10) writes "200+" and names the Free plan for people learning Python. Anaconda reserves the right to ask for proof of eligibility. These terms may change, so **read the official page yourself before using Anaconda at work**. The author did not check whether these terms apply to the conda-forge channel; the terms document does not mention it.

When to choose which? The author's recommendation for readers of this book: for general development projects (applications, automation), **venv + pip or uv** is enough, light, and free of licence questions. Consider Anaconda/conda if you work with scientific libraries that are hard to install with pip, or if your team already uses it. At work, check the licence terms above first.

🔥 TECHNICAL STUFF (feel free to skip)

On Windows, PowerShell may refuse `Activate.ps1` because of its script execution policy. The Python documentation suggests `Set-ExecutionPolicy RemoteSigned -Scope CurrentUser`. This changes a security setting of your own account, so do it only if you understand the consequences; the alternative is to call `.venv\Scripts\python.exe` directly without activation.

Exercises

1. In a terminal, make a venv `practice_venv` with `python -m venv practice_venv`, activate it, then type `python -c "import sys; print(sys.prefix)"`. Compare with the result without activation.
2. Install `six` (a small package) in that venv with `python -m pip install six`, then save `python -m pip freeze > requirements.txt` and look at its contents.
3. Change `03_read_pyproject.py` so it also prints the `dev` list, one package per line.
4. **Challenge:** with `uv`, make a new project (`uv init`), add `pytest` as a development dependency (`uv add --dev pytest`; check `uv add --help`), then run `uv run pytest --version`.

Chapter Summary

- **Environment** = a separate Python room per project; **dependency** = an outside package needed.
- **venv** makes environments; activation is optional; do not move them or put them in git, just rebuild.
- **pip** installs from PyPI; **pip freeze** produces `requirements.txt` with exact versions.
- **pyproject.toml** keeps the project identity, dependencies, and tool settings; read with `tomllib`.
- **uv** combines venv, pip, and a lock file (`uv.lock`) quickly. **poetry** is its rival (not tested here).
- **Anaconda/conda** has its own uses and licence; read the official terms before using it at work. The conda part of this book has not been tested directly.

Next, in Chapter 18: basic testing with `pytest`, how to write code that checks your own code.

Chapter 18 — Testing Basics: Checking Code Automatically with pytest

Series I1 — Tidy and Maintainable Code

In this chapter, you will:

- Understand why automatic tests save you when code changes
- Write `assert` statements and your first test function
- Run **pytest** and read the passed and failed report
- Use **fixtures** (ready-made ingredients), `parametrize`, and `pytest.raises`
- Lay out a test folder and test the `inventory` package from the running project

All scripts for this chapter are in the `code/ch18/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Imagine you change the biomass formula in the `Pond` class. Do the pen, the rice field, and the forest plot still give correct numbers? Checking everything by eye after each change is tiring, and something always slips through. The solution: write **automatic tests**, small pieces of code that call your code and compare the result with the expected answer. Like a fish scale calibrated every morning: set up once, it checks in a blink.

Tests live in their own folder and call the project code

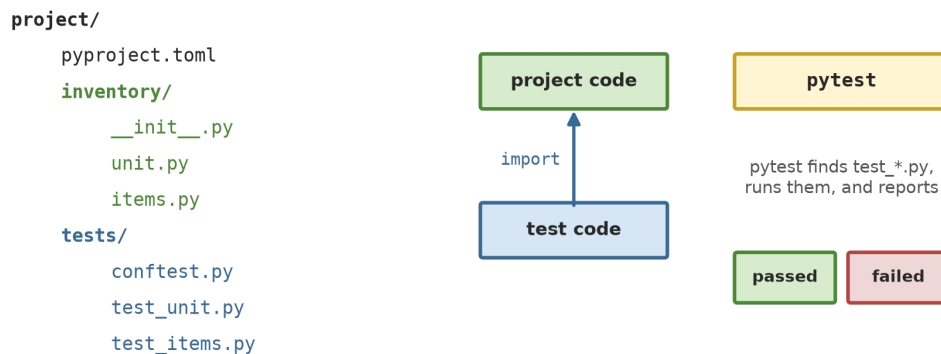


Figure 0.1: Tests live in their own folder, call the project code, and pytest reports passed or failed.

assert: “make sure this is true”

The basis of every test in Python is **assert**. If the condition is true, the program carries on silently. If false, an `AssertionError` appears (an error meaning “the statement was not satisfied”).

```
# Chapter 18 - assert: "make sure this is true, otherwise stop"  
biomass_kg = 800 * 540.5 / 1000  
  
assert biomass_kg > 0  
print("first assert passed: biomass", biomass_kg, "kg")  
  
try:  
    assert biomass_kg == 400, "biomass should be 400 kg"
```

```

except AssertionError as e:
    print("second assert failed:", e)

count = 0
try:
    assert count > 0
except AssertionError:
    print("an assert without a message can fail too")

```

```

first assert passed: biomass 432.4 kg
second assert failed: biomass should be 400 kg
an assert without a message can fail too

```

The first statement passes without a sound apart from the `print` after it. In the second, the comma adds a message that appears on failure. The script catches it with `try/except` (Chapter 12) so the program does not stop.

pytest and the first test

pytest is the most widely used test library among Python programmers. It is already installed in this book's environment; on your computer, install it with `pip install pytest` inside a venv (Chapter 17). The rules are simple:

- Test files are named **test_*.py** (or ***_test.py**).
- Test functions are named **test_...**
- Inside them, write **assert**.

(Source: the official pytest documentation, page “Good Integration Practices”, read 2026-10-10.)

```

# Chapter 18 - The first pytest test: a test_*.py file holding test_* functions
from pathlib import Path

from helpers import FOLDER, run_pytest

print(Path(FOLDER / "simple_example" / "test_calc.py").read_text(encoding="utf-8"))
print("--- pytest result ---")
text, code = run_pytest(["-v"], FOLDER / "simple_example")
print(text)
print("exit code:", code)

```

```

def biomass_kg(count, weight_g):
    return count * weight_g / 1000

def test_simple_biomass():
    assert biomass_kg(800, 500) == 400

def test_zero_biomass():
    assert biomass_kg(0, 540.5) == 0

--- pytest result ---
===== test session starts =====
collecting ... collected 2 items

test_calc.py::test_simple_biomass PASSED [ 50%]
test_calc.py::test_zero_biomass PASSED [100%]

===== 2 passed =====
exit code: 0

```

This script runs `pytest` through `subprocess` on the `simple_example/` folder. In the output, `test_calc.py` holds two test functions, both `PASSED`, and the summary says “2 passed”. Exit code `0` means everything is fine. `pytest` gives exit code `1` when any test fails, so automatic scripts and CI (Chapter 20) can use it. In this book’s output, the run time in the summary is deliberately removed so the result stays the same every time the book is built.

When a test fails

A test that always passes proves nothing yet; you need to know what the report looks like when it fails. The file `failing_example/test_failing.py` deliberately holds a wrong expectation (400 kg, while the calculation gives 432.4 kg).

```
# Chapter 18 - Reading the report of a failed test
from helpers import FOLDER, run_pytest

text, code = run_pytest(["-q"], FOLDER / "failing_example")
print(text)
print("exit code:", code)
```

```
F [100%]
===== FAILURES =====
_____ test_biomass_wrong_expectation _____

    def test_biomass_wrong_expectation():
        fish = Fish("catfish", 800, 540.5, 7.2)
>       assert fish.biomass_kg() == 400.0
E       AssertionError: assert 432.4 == 400.0
E         + where 432.4 = biomass_kg()
E         + where biomass_kg = Fish(species='catfish', count=800, avg_weight_g=540.5,
water_ph=7.2).biomass_kg

test_failing.py:6: AssertionError
===== short test summary info =====
FAILED test_failing.py::test_biomass_wrong_expectation - AssertionError: asse...
1 failed
exit code: 1
```

How to read it from the top:

- The letter **F** marks one failed test (a passed test is a dot).
- The arrow **>** points to the `assert` line that failed.
- The **E** (error) lines explain: the actual value is 432.4, the expectation 400.0. `pytest` even breaks down where 432.4 came from (`biomass_kg()` on that `Fish` object).
- Last come the `FAILED ...` summary and exit code 1.

TIP

When a test fails, do not change the test straight away to make it pass. Ask first: *is the code wrong, or the expectation?* In this example the expectation is wrong. Sometimes the code is wrong instead, and that is what tests are for.

Laying out the test folder

The `inventory` package from Chapter 16 now lives in the `project/` folder. Tests go in a `tests/` folder **next** to the package, not mixed in. `pytest` recommends this separation (official documentation, 2026-10-10).

```
project/
  pyproject.toml
  inventory/
```



```

__init__.py  unit.py  items.py  sample.py  report.py
tests/
  conftest.py
  test_items.py  test_unit.py  test_report.py

```

The pytest rules are written in `pyproject.toml` (Chapter 17):

```

[tool.pytest.ini_options]
testpaths = ["tests"]
pythonpath = ["."]

```

`testpaths` tells pytest to look for tests only in `tests/`. `pythonpath = ["."]` adds the project folder to the import path, so `import inventory` works without installing it first.

```

# Chapter 18 - The tests pytest finds in the project (without running them)
from helpers import FOLDER, run_pytest

text, code = run_pytest(["--collect-only", "-q"], FOLDER / "project")
print(text)

```

```

tests/test_items.py::test_fish_biomass
tests/test_items.py::test_tree_category[28.0-small]
tests/test_items.py::test_tree_category[32.5-medium]
tests/test_items.py::test_tree_category[45.2-large]
tests/test_items.py::test_tree_category[40.0-large]
tests/test_items.py::test_negative_fish_count_rejected
tests/test_report.py::test_save_summary
tests/test_unit.py::test_zero_area_rejected
tests/test_unit.py::test_parent_cannot_compute
tests/test_unit.py::test_pond_output
tests/test_unit.py::test_pond_summary
tests/test_unit.py::test_summary_five_units
tests/test_unit.py::test_items_cannot_be_changed_from_outside

13 tests collected

```

This is `--collect-only` mode: pytest only **collects** the tests without running them. From the three files, **13 tests** were found. The test `test_tree_category` appears four times, tagged `[28.0-small]` and so on. The next section explains why.

Fixtures, parametrize, and raises

Three of pytest's most-used tools, all present in our project.

A **fixture** is a “ready-made ingredient”: a function marked `@pytest.fixture` that is made fresh and lent to tests by name. Put it in `conftest.py` so every test file can use it.

```

"""Shared fixtures: ready-made ingredients for every test file."""

import pytest

from inventory import Fish, Pond
from inventory.sample import make_sample

@pytest.fixture
def sample_units():
    """Five sample units, freshly made for every test."""
    return make_sample()

@pytest.fixture

```

```
def pond_a():
    pond = Pond("A", 0.04)
    pond.add(Fish("catfish", 800, 540.5, 7.2))
    return pond
```

A test simply writes the fixture's name as a parameter, and pytest fills it in. Every test receives a **new** object, so tests do not disturb each other.

```
import pytest

from inventory import Fish, Tree

def test_fish_biomass():
    fish = Fish("catfish", 800, 540.5, 7.2)
    assert fish.biomass_kg() == pytest.approx(432.4)

@pytest.mark.parametrize(
    ("diameter", "expected"),
    [(28.0, "small"), (32.5, "medium"), (45.2, "large"), (40.0, "large")],
)
def test_tree_category(diameter, expected):
    assert Tree("Teak", diameter, 15.0).category() == expected

def test_negative_fish_count_rejected():
    with pytest.raises(ValueError, match="negative"):
        Fish("catfish", -1, 500.0, 7.0)
```

- **@pytest.mark.parametrize** runs one test function for several pairs of data. Four diameter pairs become four tests (that is the [28.0-small] above). The diameter 40.0 is included on purpose to test the **boundary**: the large category starts exactly at 40.
- **pytest.approx(432.4)** compares decimal numbers with a small tolerance, so rounding noise does not fail the test (remember Chapter 6: decimals can be slightly off).
- **pytest.raises(ValueError, match="negative")** states: “this code **must** raise an error”. If it does not, the test fails. This is how to test that the data guard works.

```
import pytest

from inventory import Pond, Unit

def test_zero_area_rejected():
    with pytest.raises(ValueError):
        Pond("X", 0)

def test_parent_cannot_compute():
    with pytest.raises(NotImplementedError):
        Unit("Plain", 1.0).output()

def test_pond_output(pond_a):
    assert pond_a.output() == pytest.approx(432.4)

def test_pond_summary(pond_a):
    assert pond_a.summary() == "Pond A: 1 record(s), 432.4 kg biomass"
```

```
def test_summary_five_units(sample_units):
    text = [unit.summary() for unit in sample_units]
    assert text[0] == "Forest plot A: 3 record(s), 2.0 healthy trees"
    assert text[2] == "Rice field Lowland: 1 record(s), 1250.0 kg (2500 kg/ha)"
    assert len(text) == 5

def test_items_cannot_be_changed_from_outside(pond_a):
    assert isinstance(pond_a.items, tuple)
    with pytest.raises(AttributeError):
        pond_a.items.append("x")
```

The test `test_summary_five_units` uses the `sample_units` fixture and compares the summary text exactly. There is also `test_parent_cannot_compute` for the polymorphism behaviour of Chapter 16.

```
from inventory.report import save_summary

def test_save_summary(tmp_path, sample_units):
    path = tmp_path / "summary.txt"
    count = save_summary(sample_units, path)
    lines = path.read_text(encoding="utf-8").splitlines()
    assert count == 5
    assert lines[3] == "Pond A: 1 record(s), 432.4 kg biomass"
```

The built-in fixture `tmp_path` gives a temporary folder that is thrown away after the test, so tests that write files do not litter your computer.

i REMEMBER

A good test is **small** (one thing per test), **independent** (does not rely on other tests), and **named for its expectation**. `test_zero_area_rejected` tells you at once what is broken if it fails.

Running some of the tests

In a big project you often want only the tests related to the part you are changing. The `-k` option selects tests by a piece of the name, and `-v` (*verbose*) shows each test.

```
# Chapter 18 - Running only some tests: -k selects by name
from helpers import FOLDER, run_pytest

text, code = run_pytest(["-v", "-k", "pond"], FOLDER / "project")
print(text)
```

```
===== test session starts =====
testpaths: tests
collecting ... collected 13 items / 11 deselected / 2 selected

tests/test_unit.py::test_pond_output PASSED [ 50%]
tests/test_unit.py::test_pond_summary PASSED [100%]

===== 2 passed, 11 deselected =====
```

Of the 13 tests collected, 11 were set aside (*deselected*) and only the 2 whose names contain “pond” ran.

TECHNICAL STUFF (feel free to skip)

The layout pytest suggests for a project that will be installed is the `src/` layout: the package in `src/inventory/` and tests in `tests/`, then installed with `pip install -e .` (“editable”: code changes are read at once without reinstalling). This book uses a flat layout (the package directly in `project/`) with `pythonpath = ["."]` because it is simpler for beginners. The pytest documentation also recommends `--import-mode=importlib` for new projects; that option is not used here.

Running Project I1: tests for the inventory package

Now all project tests run at once. Look back at `sample_units` from Chapter 16: those five units are now guarded by tests.

```
# Chapter 18 - Running project I1: all tests of the inventory package
from helpers import FOLDER, run_pytest

text, code = run_pytest(["-v"], FOLDER / "project")
print(text)
print("exit code:", code)
```

```
===== test session starts =====
testpaths: tests
collecting ... collected 13 items

tests/test_items.py::test_fish_biomass PASSED [ 7%]
tests/test_items.py::test_tree_category[28.0-small] PASSED [ 15%]
tests/test_items.py::test_tree_category[32.5-medium] PASSED [ 23%]
tests/test_items.py::test_tree_category[45.2-large] PASSED [ 30%]
tests/test_items.py::test_tree_category[40.0-large] PASSED [ 38%]
tests/test_items.py::test_negative_fish_count_rejected PASSED [ 46%]
tests/test_report.py::test_save_summary PASSED [ 53%]
tests/test_unit.py::test_zero_area_rejected PASSED [ 61%]
tests/test_unit.py::test_parent_cannot_compute PASSED [ 69%]
tests/test_unit.py::test_pond_output PASSED [ 76%]
tests/test_unit.py::test_pond_summary PASSED [ 84%]
tests/test_unit.py::test_summary_five_units PASSED [ 92%]
tests/test_unit.py::test_items_cannot_be_changed_from_outside PASSED [100%]

===== 13 passed =====
exit code: 0
```

All **13 tests pass**. From now on, whenever you change the `Unit` class or its children, run `python -m pytest`. If any number changes by accident, pytest tells you exactly where. That is what “safe to change” means in the closing of the series.

Exercises

1. In `simple_example/test_calc.py`, add a third test: `biomass_kg(1250, 480)` must be 600. Run it with `python -m pytest` in that folder.
2. Change the expected number in `failing_example/test_failing.py` to `pytest.approx(432.4)` and run it; make sure the test passes.
3. Write a test `test_pen_output` for the `Pen` class: add three `Livestock` records (350, 412.5, and 389.0 kg) and check the output is 1151.5.
4. **Challenge:** write a parametrized test for `Tree.category()` with five diameters, including the boundary 30.0. Then deliberately change `>= 30` to `> 30` in `items.py`, run the tests, see the boundary test fail, and restore the code.

Chapter Summary

- **Automatic tests** check your code every time they run, so changes do not break things silently.
- **assert** is the basis; files `test_*.py` holding functions `test_*` are found automatically by **pytest**.
- A failed test shows **F**; the `>` line points at the assert, the **E** lines give the actual value and the expectation. Exit code 1.
- **Fixtures** (`conftest.py`), **parametrize**, **pytest.approx**, **pytest.raises**, and **tmp_path** are everyday tools.
- Put tests in a `tests/` folder next to the package; configure under `[tool.pytest.ini_options]` in `pyproject.toml`.
- The `inventory` package now has 13 passing tests.

Next, in Chapter 19: code quality, how to tidy your writing style automatically with `ruff`, record a program's progress with logging, and document the project.

Chapter 19 — Code Quality: Tidy, Logged, and Documented

Series I1 — Tidy and Maintainable Code

In this chapter, you will:

- Meet **PEP 8**, the style guide for Python
- Use **ruff** to inspect (*linter*) and tidy (*formatter*) code automatically
- Install **pre-commit** so checks run by themselves before code is saved
- Record what a program does with **logging**, instead of `print`
- Get to know debugging in VS Code and Jupyter, and project documentation with MkDocs and Sphinx

All scripts for this chapter are in the `code/ch19/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Code is **read** more often than it is written: by colleagues, and by you six months from now. Picture a field notebook. If each officer writes with a layout of their own, finding one number becomes torture. Uniform, tidy code is easy to read, and silly mistakes show up sooner. Chapter 18 keeps code **correct**; this chapter keeps it **tidy, logged, and understandable**.

A workflow that guards code quality before saving

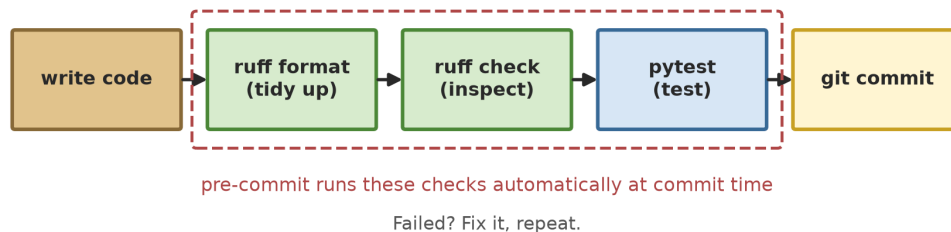


Figure 0.1: The quality workflow before code is saved: tidy up, inspect, test, then commit.

PEP 8: the grammar of Python code

PEP 8 is the official Python style guide (PEP = *Python Enhancement Proposal*). A summary from its official document (peps.python.org/pep-0008, read 2026-10-10):

Matter	PEP 8 rule
Indentation	4 spaces per level
Line length	at most 79 characters (docstrings and comments 72)
Function and variable names	lowercase with underscores: <code>calc_biomass</code>
Class names	each word capitalised (<i>CapWords</i>): <code>PondFish</code>
Constants	all capitals: <code>PH_LIMIT</code>

Matter	PEP 8 rule
Imports	at the top of the file, one per line; grouped: standard library, outside packages, your own code
Whitespace	one space around <code>=</code> , <code>==</code> , <code>and</code> ; two blank lines between functions and classes

Remember the principle, not the numbers: **consistent and easy to read**. Luckily, you need not memorise anything. A tool checks it.

ruff: inspector and tidier in one

ruff is a Python tool (made by Astral, the maker of uv) that combines two jobs: a **linter** (an inspector that looks for mistakes and bad habits) and a **formatter** (a tidier that arranges the appearance). It is already installed in this book's environment; on your computer: `python -m pip install ruff`. Let us feed ruff a file made messy on purpose:

```
# Chapter 19 - ruff check: finding problems in code (a linter)
from helpers import FOLDER, ruff

examples = FOLDER / "examples"
print((examples / "messy.py").read_text(encoding="utf-8"))
print("--- ruff check ---")
text, code = ruff("check", "--output-format", "concise", "messy.py", folder=examples)
print(text)
print("exit code:", code)
```

```
import os, sys
import json

def calc_biomass(count,weight_g,history=[]):
    l = count*weight_g/1000
    unused = 5
    if count == True:
        print( "odd count" )
    history.append(l)
    print("Biomass: %s kg" % l)
    return l

class pond_fish:
    def __init__(self,name): self.name=name

--- ruff check ---
messy.py:1:1: E401 [*] Multiple imports on one line
messy.py:1:1: I001 [*] Import block is un-sorted or un-formatted
messy.py:1:8: F401 [*] `os` imported but unused
messy.py:1:12: F401 [*] `sys` imported but unused
messy.py:2:8: F401 [*] `json` imported but unused
messy.py:4:41: B006 Do not use mutable data structures for argument defaults
messy.py:5:5: E741 Ambiguous variable name: `l`
messy.py:6:5: F841 Local variable `unused` is assigned to but never used
messy.py:7:8: E712 Avoid equality comparisons to `True`; use `count:` for truth checks
messy.py:10:11: UP031 Use format specifiers instead of percent format
messy.py:13:7: N801 Class name `pond_fish` should use CapWords convention
Found 11 errors.
[*] 5 fixable with `--fix` (4 unsafe fixes available with `--fix --unsafe-fixes`).
exit code: 1
```

Each report line has the form `file:line:column: CODE message`. A few that appear:


```

class pond_fish:
-     def __init__(self,name): self.name=name
+     def __init__(self, name):
+         self.name = name

```

1 file would be reformatted

exit code: 1

A - is an old line, a + a new one. The changes are pure appearance: a space after commas, two blank lines between functions, the body of `__init__` moved onto its own line. So the program **does not change**, only the way it is written. Exit code 1 here means “something would be changed”.

ruff check --fix: automatic repairs

```

# Chapter 19 - ruff check --fix then ruff format, on a copy of the file
import shutil

from helpers import FOLDER, ruff

work = FOLDER / "_tmp" / "fix"
shutil.rmtree(work, ignore_errors=True)
shutil.copytree(FOLDER / "examples", work)

text, _ = ruff("check", "--fix", "--output-format", "concise", "messy.py", folder=work)
print(text)
print("--- ruff format ---")
text, _ = ruff("format", "messy.py", folder=work)
print(text)
print("--- the file after tidying ---")
print((work / "messy.py").read_text(encoding="utf-8"))

```

```

messy.py:2:41: B006 Do not use mutable data structures for argument defaults
messy.py:3:5: E741 Ambiguous variable name: `l`
messy.py:4:5: F841 Local variable `unused` is assigned to but never used
messy.py:5:8: E712 Avoid equality comparisons to `True`; use `count:` for truth checks
messy.py:8:11: UP031 Use format specifiers instead of percent format
messy.py:11:7: N801 Class name `pond_fish` should use CapWords convention
Found 10 errors (4 fixed, 6 remaining).
4 unsafe fixes available with `--fix --unsafe-fixes`.
--- ruff format ---
1 file reformatted
--- the file after tidying ---
def calc_biomass(count, weight_g, history=[]):
    l = count * weight_g / 1000
    unused = 5
    if count == True:
        print("odd count")
    history.append(l)
    print("Biomass: %s kg" % l)
    return l

class pond_fish:
    def __init__(self, name):
        self.name = name

```

Notice what happened. `--fix` runs only **safe** repairs: unused imports are removed, and the import order is tidied (in the resulting file, the import lines at the top are gone). Six problems **remain**, and ruff notes that four “unsafe fixes” exist which only run if you ask with `--unsafe-fixes`. The reason: changing

`history=[]` or `== True` would **change the program's behaviour**, and that decision is yours, not the tool's. Then `ruff format` tidies the appearance.

The rest is fixed by hand. Here is the final result, checked again:

```
# Chapter 19 - The remaining problems fixed by hand (tidy.py), then checked again
from helpers import FOLDER, ruff
```

```
examples = FOLDER / "examples"
print((examples / "tidy.py").read_text(encoding="utf-8"))
print("--- ruff check ---")
print(ruff("check", "tidy.py", folder=examples)[0])
print("--- ruff format --check ---")
print(ruff("format", "--check", "tidy.py", folder=examples)[0])
```

```
def calc_biomass(count, weight_g, history=None):
    """Calculate fish biomass (kg) and record it in history."""
    if history is None:
        history = []
    biomass_kg = count * weight_g / 1000
    if count <= 0:
        print("count makes no sense")
    history.append(biomass_kg)
    print(f"Biomass: {biomass_kg} kg")
    return biomass_kg
```

```
class PondFish:
    def __init__(self, name):
        self.name = name
```

```
--- ruff check ---
All checks passed!
--- ruff format --check ---
1 file already formatted
```

What we decided ourselves: `history=None` (the list is made fresh inside the function), `l` became `biomass_kg` (a name that tells a story), the unused variable was removed, `== True` was replaced by a sensible check (`count <= 0`; the original code's intent was unclear anyway), `%` became an f-string, and `pond_fish` became `PondFish`. All checks now pass ("All checks passed!").

TIP

Make a habit of these two commands before saving your work: `ruff check --fix .` then `ruff format ..`

pre-commit: an automatic doorkeeper

You will certainly forget to run `ruff` someday. **pre-commit** is a tool that puts a "guard" on git: each time you `git commit` (save changes; covered in Chapter 20), a series of checks runs automatically. If any fails, the commit is **cancelled**. Install with `python -m pip install pre-commit`.

Its settings live in the file `.pre-commit-config.yaml`. In its official documentation (pre-commit.com, read 2026-10-10), most examples fetch hooks from repos on the internet (repo: <https://...> with rev:). So that our script does not depend on the internet, we use **local hooks** (repo: `local`) that run the `ruff` already on the computer. The script below makes a temporary git repo in `_tmp/`, installs the guard, and tries two commits.

```
# Chapter 19 - pre-commit: automatic checks at every git commit (in a temporary repo)
import os
import shutil
```

```

import subprocess
import sys
from pathlib import Path

from helpers import FOLDER, remove_folder

repo = FOLDER / "_tmp" / "repo_pre_commit"
remove_folder(repo)
repo.mkdir(parents=True)
empty_config = repo.parent / "empty_gitconfig"
empty_config.write_text("", encoding="utf-8")

# Temporary git: does not read the user's git configuration; identity via environment
variables
python_folder = str(Path(sys.executable).parent)
env = dict(
    os.environ,
    GIT_CONFIG_GLOBAL=str(empty_config), GIT_CONFIG_NOSYSTEM="1",
    GIT_AUTHOR_NAME="Test", GIT_AUTHOR_EMAIL="test@example.invalid",
    GIT_COMMITTER_NAME="Test", GIT_COMMITTER_EMAIL="test@example.invalid",
    PATH=python_folder + os.pathsep + os.environ["PATH"],    # so the 'ruff' command
is found
    PYTHONUTF8="1", NO_COLOR="1",
)

def run(*command):
    result = subprocess.run(command, cwd=repo, env=env, text=True, encoding="utf-8",
                             stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    return result.stdout.strip(), result.returncode

(repo / ".pre-commit-config.yaml").write_text("""\
repos:
- repo: local
  hooks:
    - id: ruff-check
      name: ruff check
      entry: ruff check
      language: unsupported
      types: [python]
    - id: ruff-format
      name: ruff format
      entry: ruff format --check
      language: unsupported
      types: [python]
""", encoding="utf-8")
shutil.copy(FOLDER / "examples" / "messy.py", repo / "messy.py")
shutil.copy(FOLDER / "examples" / "pyproject.toml", repo / "pyproject.toml")

run("git", "init", "--quiet", "-b", "main")
print(run(sys.executable, "-m", "pre_commit", "install")[0].replace(str(repo),
"<repo>"))

run("git", "add", ".")
text, code = run("git", "commit", "-m", "add messy.py")
print("--- first commit (messy code) ---")
print("\n".join(line for line in text.splitlines() if line.startswith(("ruff", "- hook
id", "- exit code"))))
print("commit succeeded?", code == 0)

```

```
shutil.copy(FOLDER / "examples" / "tidy.py", repo / "messy.py")
run("git", "add", ".")
text, code = run("git", "commit", "-m", "fix messy.py")
print("--- second commit (code is tidy) ---")
print("\n".join(line for line in text.splitlines() if line.startswith("ruff")))
print("commit succeeded?", code == 0)
print("--- history ---")
print(run("git", "log", "--format=%s")[0])
```

```
pre-commit installed at .git\hooks\pre-commit
--- first commit (messy code) ---
ruff check.....Failed
- hook id: ruff-check
- exit code: 1
ruff format.....Failed
- hook id: ruff-format
- exit code: 1
commit succeeded? False
--- second commit (code is tidy) ---
ruff check.....Passed
ruff format.....Passed
commit succeeded? True
--- history ---
fix messy.py
```

The storyline:

- `pre-commit install` places the guard script at `.git/hooks/pre-commit`.
- **The first commit** (messy code): both hooks answer `Failed`, and the commit **does not happen** (`False`).
- The file is replaced by the tidy version, then **the second commit**: both `Passed`, the commit succeeds, and the git history holds one entry.

The pre-commit documentation notes that the value `language: system` is now only an alias for `unsupported` (since version 4.4.0) and will be removed; new configurations should use `unsupported`, as in this script. For older pre-commit versions, change it to `system`. This temporary repo uses a git kept apart from your git configuration (see Chapter 20).

Logging: a program's diary

`print()` suits experiments, but in a real program you need something more orderly: **logging**, the built-in module for recording events with a **level of importance**. According to the Python documentation (Logging HOWTO, read 2026-10-10):

Level	Value	Use
DEBUG	10	technical detail for hunting bugs
INFO	20	confirmation that things work as expected
WARNING	30	something unexpected, the program still works
ERROR	40	a function could not be carried out
CRITICAL	50	a serious error

A logger passes on only messages whose level is **equal to or higher** than its effective level. The default level of the root logger is `WARNING`, so without configuration `INFO` and `DEBUG` do not show.

```
# Chapter 19 - logging: a program's diary with levels of importance
import logging
import sys
```

```

logging.basicConfig(
    level=logging.INFO,
    format="%(levelname)-8s | %(name)s | %(message)s",
    stream=sys.stdout,
)
log = logging.getLogger("pond")

log.debug("technical detail (not shown at level INFO)")
log.info("pond A: 800 fish stocked")
log.warning("pH 5.9 is below the 6.5 limit")
log.error("temperature sensor is not responding")

try:
    average = 540.5 / 0
except ZeroDivisionError:
    log.exception("average calculation failed")      # also prints the traceback

print("effective level:", logging.getLevelName(log.getEffectiveLevel()))

```

```

INFO      | pond | pond A: 800 fish stocked
WARNING   | pond | pH 5.9 is below the 6.5 limit
ERROR     | pond | temperature sensor is not responding
ERROR     | pond | average calculation failed
Traceback (most recent call last):
  File "05_logging_basics.py", line 18, in <module>
    average = 540.5 / 0
    ~~~~~^~
ZeroDivisionError: division by zero
effective level: INFO

```

- `basicConfig(level=..., format=..., stream=...)` sets the level and the shape of the records. The form `%(levelname)-8s` means the level name, left-aligned, 8 letters wide.
- The `DEBUG` message **does not show** because the level is set to `INFO`. The other three show.
- `log.exception(...)` is called inside an `except`: it records the message at `ERROR` level **together with the traceback**. This is the most useful replacement for `print`.
- `getEffectiveLevel()` shows the effective level: `INFO`.

Logging in the inventory package

The Python documentation suggests each module use `logger = logging.getLogger(__name__)`. Logger names then follow the package hierarchy (`inventory.unit` is a child of `inventory`), and messages rise to the parent logger. It also advises that libraries **not** install their own output handlers: that is the business of the application using them. That is what we follow: the `inventory` package only **records** (`logger.debug` in `Unit.add`, `logger.info` in `save_summary`), and the script decides where the records go.

```

# Chapter 19 - logging in the inventory package: the screen shows INFO, the file
keeps DEBUG
import logging
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent / "project"))

from inventory.report import save_summary # noqa: E402
from inventory.sample import make_sample # noqa: E402

work = Path(__file__).parent / "_tmp"
work.mkdir(exist_ok=True)
log_file = work / "inventory.log"
log_file.unlink(missing_ok=True)

```

```

log_format = "%(levelname)s %(name)s: %(message)s"
screen = logging.StreamHandler(sys.stdout)
screen.setLevel(logging.INFO)
file_handler = logging.FileHandler(log_file, encoding="utf-8")
file_handler.setLevel(logging.DEBUG)
for handler in (screen, file_handler):
    handler.setFormatter(logging.Formatter(log_format))

root = logging.getLogger("inventory")          # parent of inventory.unit and
inventory.report
root.setLevel(logging.DEBUG)
root.addHandler(screen)
root.addHandler(file_handler)

units = make_sample()
save_summary(units, work / "summary.txt")
file_handler.close()

lines = log_file.read_text(encoding="utf-8").splitlines()
print(f"Log file contents: {len(lines)} lines")
print("First line:", lines[0])
print("Last line :", lines[-1])

```

```

INFO inventory.report: writing 5 summaries to summary.txt
Log file contents: 11 lines
First line: DEBUG inventory.unit: Forest plot A received Tree
Last line : INFO inventory.report: writing 5 summaries to summary.txt

```

Two handlers are attached to the `inventory` logger: the screen takes level `INFO` and up, and the file takes everything down to `DEBUG`. The result: the screen shows just one line (`writing 5 summaries ...`), while the log file holds **11 lines**: 10 `DEBUG` records (one for each call to `add`: 3 trees + 2 + 1 + 1 + 3 = 10) plus 1 `INFO` record.

Debugging: hunting bugs carefully

Debugging means running a program while stopping at a chosen point to look inside variables. The most basic way is already in Python: the built-in function `breakpoint()` stops the program and opens the `pdb` debugger in the terminal.

```

# Chapter 19 - Where to place a breakpoint when hunting a bug
def biomass_kg(count, weight_g):
    total_g = count * weight_g
    breakpoint()          # the program stops here; type "p total_g" then "c"
    to continue
    return total_g / 1000

print(biomass_kg(800, 540.5))

```

This code was **not run** when the book was built because `breakpoint()` stops the program and waits for input from the user, which does not suit an automatic build. The author tried it once with the input `p total_g` then `c`: the program stopped at the `breakpoint()` line, `p total_g` printed 432400.0 (grams), and `c` (*continue*) carried on until it printed 432.4.

In VS Code, a more comfortable way (described from the official VS Code documentation, read 2026-10-10; **not run by the author**):

- Install the Python extension (the *Python Debugger* extension comes with it).
- Click in the editor's left margin to set a **breakpoint**. A breakpoint can have a **condition** or a hit count, for instance stopping only after five hits.

- Press **F5** (*Start Debugging*) or choose *Debug Python File*. The program stops at the breakpoint, and you can inspect variables.
- Finer settings are kept in `.vscode/launch.json` (name, the program to run, `args`, and so on).

In a **Jupyter notebook inside VS Code** (also from the official documentation, not run): **Run by Line** runs a cell one statement at a time, and **Debug Cell** gives the full debugger with breakpoints. Both need ipykernel version 6 or later.

⚠ WARNING

The debugger answers “why is the value like this **at this point**?”. Do not use it as a replacement for tests. Tests (Chapter 18) stop the same bug from coming back; the debugger only helps find it once.

Project documentation: MkDocs and Sphinx

Code without usage notes is soon forgotten. Two popular tools turn text files into a documentation site:

- **MkDocs** (mkdocs.org): write pages in **Markdown**, then `mkdocs build` produces a static HTML site in the `site/` folder. `mkdocs serve` opens a local preview. The only required setting in `mkdocs.yml` is `site_name`; `nav` sets the menu order (official documentation, read 2026-10-10).
- **Sphinx** (sphinx-doc.org): an older and richer tool, using **reStructuredText**. `sphinx-quickstart` makes the skeleton (`conf.py`, `index.rst`), `make html` builds the site, and the **autodoc** extension pulls documentation straight from docstrings (official documentation, read 2026-10-10). Sphinx is not installed or run in this chapter.

The script below builds a real mini MkDocs site:

```
# Chapter 19 - MkDocs: documentation pages from Markdown files (builds a real site
in _tmp)
import shutil
import sys

from helpers import FOLDER, run

site = FOLDER / "_tmp" / "docs_site"
shutil.rmtree(site, ignore_errors=True)
(site / "docs").mkdir(parents=True)

(site / "mkdocs.yml").write_text("""\
site_name: Inventory Documentation
nav:
  - Home: index.md
  - How to use: how-to-use.md
""", encoding="utf-8")
(site / "docs" / "index.md").write_text(
    "# Inventory\n\nSample package for recording forest, garden, rice field, pond, and
pen.\n",
    encoding="utf-8")
(site / "docs" / "how-to-use.md").write_text(
    "# How to use\n\n``python\nfrom inventory.sample import make_sample\n``\n",
    encoding="utf-8")

text, code = run([sys.executable, "-m", "mkdocs", "build", "--quiet"], site)
print("exit code:", code, "| message:", text or "(none)")

pages = sorted(p.relative_to(site / "site").as_posix() for p in (site /
"site").rglob("*.html"))
print("HTML pages produced:", pages)
html = (site / "site" / "index.html").read_text(encoding="utf-8")
print("Site title is on the page?", "Inventory Documentation" in html)
```

```
exit code: 0 | message: (none)
HTML pages produced: ['404.html', 'how-to-use/index.html', 'index.html']
Site title is on the page? True
```

Exit code 0 with no message means the build went fine. Three HTML pages were produced: `index.html` and `how-to-use/index.html` from our two Markdown files, plus `404.html`, which MkDocs makes itself. The site title from `mkdocs.yml` is indeed on the page.

i REMEMBER

The **docstrings** you have been writing since Chapter 10 are the cheapest documentation: they are read by `help()`, by editors, and by tools such as Sphinx autodoc. A one-sentence docstring on every function and class already helps a lot.

Running Project I1: tidying inventory

Now all the tools are used on the `inventory` package, which now has logging and 14 tests (one new test checks the log record with the `caplog` fixture). The script copies the project into `_tmp/`, checks it, repairs it, and tests it again.

```
# Chapter 19 - Running project I1: tidying the inventory package with ruff, then
testing again
import re
import shutil
import sys

from helpers import FOLDER, ruff, run

work = FOLDER / "_tmp" / "project_tidy"
shutil.rmtree(work, ignore_errors=True)
shutil.copytree(FOLDER, work, ignore=shutil.ignore_patterns("__pycache__"),
                "project",
                work, ignore=shutil.ignore_patterns("__pycache__"))

def check(title):
    text, _ = ruff("check", "--output-format", "concise", ".", folder=work)
    print(f"--- {title} ---")
    print(text)

check("ruff check, before")

ruff("check", "--fix", "--quiet", ".", folder=work)    # safe automatic fixes (import
order)
print("--- ruff format ---")
print(ruff("format", ".", folder=work)[0])
check("ruff check, after the automatic fixes")

# One docstring is too long: ruff will not guess your sentence, so shorten it by hand
fixes = {
    "inventory/__init__.py": (
        "Package inventory: information system for forest, garden, rice field, pond,
and pen (fictional data).",
        "Package inventory: fictional data for forest, garden, rice field, pond,
and pen.",
    ),
}
for path, (old, new) in fixes.items():
    file = work / path
```



```

        file.write_text(file.read_text(encoding="utf-8").replace(old, new),
encoding="utf-8")
check("ruff check, after the manual fix")
print(ruff("format", "--check", ".", folder=work)[0])

text, code = run([sys.executable, "-m", "pytest", "-q", "-p", "no:cacheprovider"],
work)
print("--- pytest ---")
print(re.sub(r" in \d+\.\d+s", "", text.splitlines()[-1]))

```

```

--- ruff check, before ---
inventory\__init__.py:1:89: E501 Line too long (107 > 88)
inventory\sample.py:3:1: I001 [*] Import block is un-sorted or un-formatted
inventory\sample.py:3:89: E501 Line too long (91 > 88)
Found 3 errors.
[*] 1 fixable with `--fix`.
--- ruff format ---
3 files reformatted, 6 files left unchanged
--- ruff check, after the automatic fixes ---
inventory\__init__.py:1:89: E501 Line too long (107 > 88)
Found 1 error.
--- ruff check, after the manual fix ---
All checks passed!
9 files already formatted
--- pytest ---
14 passed

```

Trace it:

- **Before:** 3 findings: two **E501** (lines over 88 characters) and one **I001** (import order).
- **--fix** and **format** solved two findings (imports tidied, the long line wrapped). The remaining **1** is a docstring that is too long: ruff will not guess your sentence, so we shorten it by hand.
- Afterwards all checks pass, 9 files are tidy, and **all 14 tests still pass**: tidying did not break behaviour. That is what the tests of Chapter 18 are for.

Exercises

1. Run **ruff check** on an old script of yours (for instance from Chapter 10). How many findings? Pick one code and look up its meaning with **python -m ruff rule F841**.
2. Change **05_logging_basics.py** so the level is **DEBUG**. Which message is added?
3. Add a **logger.warning(...)** to **Pond** when **water_ph** is outside 6.5 to 8.5, then watch it through **06_logging_project.py**. (These limits are only example numbers.)
4. **Challenge:** change **07_pre_commit.py** so the first file holds only **import os** and one tidy function. Which hook fails, and which passes? Explain why.

Chapter Summary

- **PEP 8** is the official Python style guide: consistent and easy to read.
- **ruff check** inspects (codes **E**, **F**, **I**, **B**, ...), **--fix** repairs only what is safe, **ruff format** arranges the appearance without changing behaviour.
- **pre-commit** runs automatic checks at every **git commit** and cancels the commit if they fail.
- **logging** records with levels **DEBUG**, **INFO**, **WARNING**, **ERROR**, **CRITICAL**; modules use **getLogger(__name__)**, and the application picks where records go.
- **Debugging:** **breakpoint()** (pdb) or the VS Code and Jupyter features; hunting bugs, not a replacement for tests.
- **MkDocs** (Markdown) and **Sphinx** (reStructuredText) make documentation sites from text files and docstrings.

Next, in Chapter 20: git, GitHub, and the project workflow: saving history, branching, working together, and closing Series II.

Chapter 20 — Git, GitHub, and the Project Workflow: A Time Machine for Your Code

Series I1 — Tidy and Maintainable Code

In this chapter, you will:

- Understand **git**: saving version history with **commits**, seeing differences, and undoing changes
- Work in **branches**, join them with **merge**, and resolve **conflicts**
- Keep secrets out of the repo with **.gitignore**
- Meet **GitHub**: repos, issues, pull requests, and reviews, plus basic **GitHub Actions**
- Close Series I1: the inventory project becomes classes, tested, tidy, and kept in git

All scripts for this chapter are in the `code/ch20/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Have you ever saved `report_final.docx`, then `report_final2.docx`, then `report_final_CORRECT.docx`? That is the manual way to track versions, and it gets messy fast. **Git** is a tool that does it neatly: it records every change as **history**, so you can see who changed what and when, and go back to an old state if something goes wrong. Picture a garden ledger: every correction is written down with its date, never erased.

Safety rules of this chapter. Every git command in this book runs in a **temporary repo** in the `_tmp/` folder, with settings kept apart from your git configuration. No GitHub repo is created, nothing is pushed, and no account is used. The GitHub part is only explained.

Git branches: work on a side track, then merge

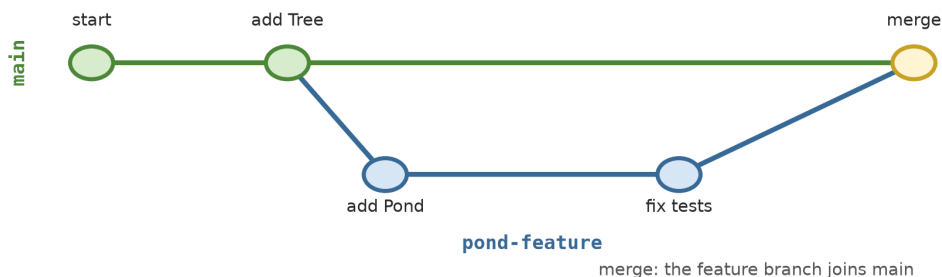


Figure 0.1: Git branches: a side track, then joined back to the main track.

Basic git concepts

Term	Meaning
repo (<i>repository</i>)	a project folder whose history git tracks (it holds a hidden <code>.git</code> folder)
commit	one “photo” of the project’s state, with a message about what changed
staging (<code>git add</code>)	choosing which changes go into the next commit

Term	Meaning
branch	a separate line of history; the main branch is usually called <code>main</code>
merge	joining another branch into the branch that is active
remote	a copy of the repo somewhere else, for instance GitHub

The flow: you **change files**, choose them with `git add`, then `git commit` to save them into history. The command `git status` tells you where you stand. (Git records the name and email of the commit's author. On your computer, set them once with `git config --global user.name "Your Name"` and `git config --global user.email "you@example.com"`. This book's scripts do not touch those settings: their test identity is set through environment variables.)

```
# Chapter 20 - git: a new repo, the first file, the first commit (in a temporary repo)
from helpers import git, new_repo, write

repo = new_repo("basic_repo")
write(repo, "harvest.csv", "block,yield_kg\nA,245.5\nB,310.0\n")

print("Initial status:", git("status", "--short", repo=repo)[0])
git("add", "harvest.csv", repo=repo)
print("After add      :", git("status", "--short", repo=repo)[0])
git("commit", "--quiet", "-m", "Record harvest of blocks A and B", repo=repo)
print("After commit  :", repr(git("status", "--short", repo=repo)[0]))

print("--- git log ---")
print(git("log", "--format=%an: %s", repo=repo)[0])
```

```
Initial status: ?? harvest.csv
After add      : A  harvest.csv
After commit   : ''
--- git log ---
Test: Record harvest of blocks A and B
```

Trace the three status lines: `??` means the file is **not yet tracked**, `A` means it is **added** (waiting for a commit), and empty text after the commit means **no pending changes**. The command `git log --format=...` shows the history: one commit with its message.

Seeing differences and undoing

```
# Chapter 20 - Changing a file, seeing the difference (diff), and undoing changes
from helpers import git, new_repo, write

repo = new_repo("diff_repo")
write(repo, "harvest.csv", "block,yield_kg\nA,245.5\nB,310.0\n")
git("add", ".", repo=repo)
git("commit", "--quiet", "-m", "Record harvest of blocks A and B", repo=repo)

write(repo, "harvest.csv", "block,yield_kg\nA,245.5\nB,312.0\nC,198.5\n")
print("Status:", git("status", "--short", repo=repo)[0])

text, _ = git("diff", repo=repo)
print("--- git diff ---")
print("\n".join(line for line in text.splitlines() if not line.startswith("index ")))

git("commit", "--quiet", "-a", "-m", "Correct block B and add block C", repo=repo)
```

```

print("--- history ---")
print(git("log", "--format=%s", repo=repo)[0])

write(repo, "harvest.csv", "typing mistake\n")
git("restore", "harvest.csv", repo=repo)          # throws away uncommitted changes
print("--- after git restore ---")
print((repo / "harvest.csv").read_text(encoding="utf-8"), end="")

```

```

Status: M harvest.csv
--- git diff ---
diff --git a/harvest.csv b/harvest.csv
--- a/harvest.csv
+++ b/harvest.csv
@@ -1,3 +1,4 @@
 block,yield_kg
 A,245.5
-B,310.0
+B,312.0
+C,198.5
--- history ---
Correct block B and add block C
Record harvest of blocks A and B
--- after git restore ---
block,yield_kg
A,245.5
B,312.0
C,198.5

```

- M (a space then M) means the file has been **modified** but not yet added.
- **git diff** shows the difference: lines starting with - were removed, + added. Here B,310.0 became B,312.0 and the block C line was added. (The technical line starting with index is left out of the display.)
- **git commit -a** saves all changed files **that are already tracked**, without a separate **git add**.
- **git restore file** throws away uncommitted changes. The end result: the file returns to the last commit's contents, so the "typing mistake" is gone. Careful: changes thrown away this way **cannot be recovered**.

Branch and merge

A **branch** lets you work on a new feature on a side track without disturbing **main**. If it works, the track is joined. If it fails, just throw it away.

```

# Chapter 20 - Branches and merge: work on a side track, then join
from helpers import git, new_repo, tidy, write

repo = new_repo("branch_repo")
write(repo, "units.txt", "forest plot\ngarden\n")
git("add", ".", repo=repo)
git("commit", "--quiet", "-m", "Initial list of units", repo=repo)

git("switch", "--quiet", "-c", "pond-feature", repo=repo)
print("Active branch:", git("branch", "--show-current", repo=repo)[0])
write(repo, "pond.txt", "pond A: 800 fish\n")
git("add", ".", repo=repo)
git("commit", "--quiet", "-m", "Add pond data", repo=repo)

git("switch", "--quiet", "main", repo=repo)
print("On main, does pond.txt exist?", (repo / "pond.txt").exists())

write(repo, "units.txt", "forest plot\ngarden\npen\n")

```

```
git("commit", "--quiet", "-a", "-m", "Add pen", repo=repo)

text, _ = git("merge", "--no-edit", "pond-feature", repo=repo)
print("--- git merge ---")
print(text.splitlines()[0])
print("On main, does pond.txt exist?", (repo / "pond.txt").exists())
print("--- git log --graph ---")
print(tidy(git("log", "--graph", "--format=%s", repo=repo)[0]))
```

```
Active branch: pond-feature
On main, does pond.txt exist? False
--- git merge ---
Merge made by the 'ort' strategy.
On main, does pond.txt exist? True
--- git log --graph ---
* Merge branch 'pond-feature'
| \
| * Add pond data
* | Add pen
| /
* Initial list of units
```

Note the interesting part: before the merge, `pond.txt` **does not exist** on `main` (it lives on the `pond-feature` branch). After the merge, it does. The `git log --graph` diagram shows the shape: two tracks (the * on the left and the * on the right) that split from the commit “Initial list of units” and meet at the merge commit. The change on `main` (“Add pen”) and on the branch (“Add pond data”) touch different files, so git joins them by itself.

Conflict: when two people change the same line

What if both tracks change the **same line**? Git does not dare to choose; it stops and asks a human to decide. This is called a **conflict**, and it is not a sign you did wrong but an ordinary thing in teamwork.

```
# Chapter 20 - Merge conflict: two branches change the same line
from helpers import git, new_repo, tidy, write

repo = new_repo("conflict_repo")
write(repo, "pond.txt", "pond A\naverage weight: 500 g\n")
git("add", ".", repo=repo)
git("commit", "--quiet", "-m", "Initial pond data", repo=repo)

git("switch", "--quiet", "-c", "weighing-fix", repo=repo)
write(repo, "pond.txt", "pond A\naverage weight: 540 g\n")
git("commit", "--quiet", "-a", "-m", "Weight from re-weighing: 540 g", repo=repo)

git("switch", "--quiet", "main", repo=repo)
write(repo, "pond.txt", "pond A\naverage weight: 520 g\n")
git("commit", "--quiet", "-a", "-m", "Weight from old notes: 520 g", repo=repo)

text, code = git("merge", "--no-edit", "weighing-fix", repo=repo)
print("--- git merge ---")
print(text)
print("exit code:", code)
print("--- contents of pond.txt during the conflict ---")
print((repo / "pond.txt").read_text(encoding="utf-8"), end="")
print("--- status ---")
print(git("status", "--short", repo=repo)[0])

# Resolution: a human decides which number is right, then removes the markers
write(repo, "pond.txt", "pond A\naverage weight: 540 g\n")
```

```
git("add", "pond.txt", repo=repo)
git("commit", "--quiet", "--no-edit", repo=repo)
print("--- after resolving ---")
print(tidy(git("log", "--graph", "--format=%s", repo=repo)[0]))
```

```
--- git merge ---
Auto-merging pond.txt
CONFLICT (content): Merge conflict in pond.txt
Automatic merge failed; fix conflicts and then commit the result.
exit code: 1
--- contents of pond.txt during the conflict ---
pond A
<<<<<<< HEAD
average weight: 520 g
=====
average weight: 540 g
>>>>>>> weighing-fix
--- status ---
UU pond.txt
--- after resolving ---
* Merge branch 'weighing-fix'
|\
| * Weight from re-weighing: 540 g
* | Weight from old notes: 520 g
|/
* Initial pond data
```

How to read the conflicted file:

- <<<<<<< HEAD to ===== is the version on **your current** branch (main: 520 g from old notes).
- ===== to >>>>>>> weighing-fix is the version from the branch being merged (540 g from re-weighing).
- The status UU means both sides changed this file and it is not yet resolved.

To resolve it: **edit the file**, remove the three marker lines, choose (or combine) the right content, then `git add` and `git commit`. The script picks 540 g because that is the re-weighing result. The decision of **which number is right** can only be made by a human who knows the reality in the field. The final history shows the merge commit that joins the two.

.gitignore and guarding secrets

Not every file belongs in git: the `.venv` folder (Chapter 17), temporary files, big downloads, and above all **secrets** such as passwords and API keys. The list git ignores is written in the `.gitignore` file. Secrets are best kept in an ignored `.env` file, or in environment variables, and read by the program through `os.environ`.

```
# Chapter 20 - .gitignore and keeping secrets out of the repo (the key value here is
only a stand-in)
from helpers import git, new_repo, write

repo = new_repo("secret_repo")
write(repo, ".env", "API_KEY=[redacted]\n")
write(repo, "app.py", "import os\nkey = os.environ['API_KEY']\n")
write(repo, "_tmp/result.txt", "temporary file\n")

print("--- without .gitignore ---")
print(git("status", "--short", repo=repo)[0])

write(repo, ".gitignore", ".env\n_tmp/\n__pycache__/\n.venv/\n")
print("--- with .gitignore ---")
print(git("status", "--short", repo=repo)[0])
```

```

print("Why is .env ignored?", git("check-ignore", "-v", ".env", repo=repo)[0])

git("add", ".", repo=repo)
git("commit", "--quiet", "-m", "Start of project", repo=repo)
print("--- files tracked by git ---")
print(git("ls-files", repo=repo)[0])

print("=== A wrong step: the secret was already committed ===")
repo2 = new_repo("committed_repo")
write(repo2, ".env", "API_KEY=[redacted]\n")
git("add", ".", repo=repo2)
git("commit", "--quiet", "-m", "oops", repo=repo2)
write(repo2, ".gitignore", ".env\n")
git("add", ".gitignore", repo=repo2)
git("commit", "--quiet", "-m", "add .gitignore", repo=repo2)
print("Tracked after .gitignore was added:", git("ls-files", repo=repo2)[0].split())
print("The first commit still contains .env:", ".env" in git("show", "--stat", "--format=", "HEAD~1", repo=repo2)[0])
git("rm", "--cached", "--quiet", ".env", repo=repo2)
git("commit", "--quiet", "-m", "stop tracking .env", repo=repo2)
print("Tracked after git rm --cached:", git("ls-files", repo=repo2)[0].split())
print("But old history still contains it:", ".env" in git("show", "--stat", "--format=", "HEAD~2", repo=repo2)[0])

```

```

--- without .gitignore ---
?? .env
?? _tmp/
?? app.py
--- with .gitignore ---
?? .gitignore
?? app.py
Why is .env ignored? .gitignore:1:.env .env
--- files tracked by git ---
.gitignore
app.py
=== A wrong step: the secret was already committed ===
Tracked after .gitignore was added: ['.env', '.gitignore']
The first commit still contains .env: True
Tracked after git rm --cached: ['.gitignore']
But old history still contains it: True

```

Part one: without `.gitignore`, git offers `.env` and `_tmp/` for adding (??). With `.gitignore`, both vanish from the list. `git check-ignore -v` explains **which rule** applies: line 1 of `.gitignore`. Only `.gitignore` and `app.py` are tracked.

Part two is the **most dangerous trap**. If a secret was **already committed**, adding it to `.gitignore` is **not enough**: the file stays tracked (`['.env', '.gitignore']`). `git rm --cached` stops the tracking, but **the old history still holds its contents** (True). Anyone holding a copy of the repo can still read it.

⚠ WARNING

A secret that has ever entered a commit must be treated as **leaked**, especially if the repo was ever sent to GitHub. The safest step (the author's advice): **revoke and replace the key** at the service provider, then clean the history. Prevention is far easier than repair: make `.gitignore` **before** the first commit. GitHub's documentation also describes *push protection*, a feature that blocks a push when a secret is detected; its availability depends on account and repo settings (read 2026-10-10), so do not make it your only fence.

GitHub: git for working together

GitHub is a website service that keeps git repos on the internet (git and GitHub are two different things: git is the tool, GitHub is one place to keep it). This part is explanation only; **nothing is run**, because creating a repo and sending code changes data on the internet and needs an account.

- A **repo on GitHub** is connected to the local repo as a **remote** (usually called **origin**). `git push` sends commits there, `git pull` fetches other people's changes.
- An **issue** is a note about a task, idea, or bug in a repo. According to the GitHub documentation (read 2026-10-10), issues can carry labels and assignees, and closing keywords in a pull request can close the issue automatically when it is merged.
- A **pull request** (PR) is a proposal to merge changes into the project. The *Files changed* tab shows the difference being reviewed, the *Conversation* tab holds discussion and reviews, and the merge status shows what still blocks it, such as missing approvals.
- **Review**: a colleague reads your changes in the PR and comments, then you fix them before merging.

```
# Chapter 20 - Commands to connect a local repo to GitHub (an example only; replace
YOUR-NAME with your account name)
import subprocess

commands = [
    ["git", "remote", "add", "origin", "https://github.com/YOUR-NAME/inventory.git"],
    ["git", "push", "-u", "origin", "main"],          # send the main branch
to GitHub
    ["git", "switch", "-c", "pond-feature"],          # work on a new branch
    ["git", "push", "-u", "origin", "pond-feature"],  # send the feature branch
    ["gh", "pr", "create", "--fill"],                 # open a pull request (GitHub
CLI tool)
]
for c in commands:
    print("$", " ".join(c))
    subprocess.run(c, check=True)
```

This code was **not run** when the book was built because creating a repo and pushing code to GitHub needs a GitHub account and changes data on the internet, so no output is shown. The command `gh pr create` belongs to the GitHub CLI; its existence was checked through `gh pr create --help`, but it was not run.

The daily workflow

Even a small team can use this habit:

1. **Fetch** the latest version (`git pull`).
2. **Make a branch** for one job (`git switch -c feature-name`).
3. **Change the code**, then run `ruff check --fix .`, `ruff format .`, and `python -m pytest` (Chapters 18 and 19). `pre-commit` can do it automatically at commit time.
4. **Commit** with a message that explains *what and why*, not just “fix”.
5. **Push** the branch, open a **pull request**, ask for **review**.
6. Once approved and all checks are green, **merge** into **main**.

The habit “one branch, one purpose, small commits” keeps conflicts rare and history easy to read.

GitHub Actions: automatic checks on a server

Checks on your own computer can be skipped. **GitHub Actions** runs automatic checks on GitHub's servers at every push or pull request. The flow is written in a YAML file in the `.github/workflows/` folder (GitHub documentation, read 2026-10-10). Here is an example for our project:

```
name: Tests

on:
```



```

push:
  branches: [main]
pull_request:

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Check out code
        uses: actions/checkout@v6
      - name: Set up Python
        uses: actions/setup-python@v7
        with:
          python-version: "3.12"
      - name: Install package and tools
        run: python -m pip install -e ".[dev]"
      - name: Check code style
        run: |
          ruff check .
          ruff format --check .
      - name: Run tests
        run: python -m pytest -q

```

- **on:** sets the **triggers**: every push to `main` and every pull request.
- **jobs:** holds the jobs; `runs-on: ubuntu-latest` picks a GitHub machine; **steps:** are the ordered steps. `uses:` applies a ready-made action, `run:` runs a command.
- The action version numbers (`@v6`, `@v7`) were taken from the documentation on 2026-10-10 and may change; check the documentation when you use them.

This workflow was **not run** by GitHub when the book was built (there is no GitHub repo), so it is **not proven to work**. The author only checked its **YAML syntax**, which is all that can be proven here:

```

# Chapter 20 - Checking the syntax of a GitHub Actions workflow file (YAML syntax only;
# GitHub did not run it)
from pathlib import Path

import yaml

file = Path(__file__).parent / "example" / ".github" / "workflows" / "test.yml"
workflow = yaml.safe_load(file.read_text(encoding="utf-8"))

print("Workflow name:", workflow["name"])
print("Triggers (on):", sorted(workflow[True]))          # YAML reads the word 'on'
as True
for job_name, job in workflow["jobs"].items():
    print(f"Job '{job_name}' runs on {job['runs-on']} with {len(job['steps'])} steps:")
    for step in job["steps"]:
        print("  -", step["name"])

```

```

Workflow name: Tests
Triggers (on): ['pull_request', 'push']
Job 'test' runs on ubuntu-latest with 5 steps:
  - Check out code
  - Set up Python
  - Install package and tools
  - Check code style
  - Run tests

```

The script uses the PyYAML library (`python -m pip install pyyaml`). The result shows the file reads correctly: one job holding five steps. A technical note: YAML reads the word `on` as `True`, which is why the key is written `workflow[True]` in the script.

Running Project I1: everything together

This is the summit of the series. The `inventory` package (classes from Chapter 16, installed in Chapter 17, tested in Chapter 18, tidied in Chapter 19) now lives in a git repo with a healthy workflow: checks, a branch, a merge.

```
# Chapter 20 - Running project I1: classes + tests + ruff + git in one workflow
import re
import shutil
import sys

from helpers import FOLDER, git, new_repo, run, tidy, write

repo = new_repo("project_i1")
shutil.copytree(FOLDER / "final_project", repo, dirs_exist_ok=True,
                ignore=shutil.ignore_patterns("__pycache__"))
write(repo, ".gitignore", ".venv/\n__pycache__/\n.pytest_cache/\n.ruff_cache/\n.env\n")

def checks():
    """The three checks pre-commit would also run: ruff check, ruff format, pytest."""
    ruff = [sys.executable, "-m", "ruff"]
    a = run([*ruff, "check", "--no-cache", "."], repo)[0].splitlines()[-1]
    b = run([*ruff, "format", "--no-cache", "--check", "."], repo)[0].splitlines()[-1]
    c = run([sys.executable, "-m", "pytest", "-q", "-p", "no:cacheprovider"], repo)[0]
    print("  ruff check :", a)
    print("  ruff format:", b)
    print("  pytest      :", re.sub(r" in \d+\.\d+s", "", c.splitlines()[-1]))

print("1. Checks on main")
checks()
git("add", ".", repo=repo)
git("commit", "--quiet", "-m", "Package inventory: classes, tests, and configuration",
    repo=repo)

print("2. New branch: a test for RiceField")
git("switch", "--quiet", "-c", "rice-field-test", repo=repo)
write(repo, "tests/test_rice_field.py", '''import pytest

from inventory import Crop, RiceField

def test_rice_field_productivity():
    rice_field = RiceField("Lowland", 0.5)
    rice_field.add(Crop("rice", "1", 110, 1250.0))
    assert rice_field.productivity() == pytest.approx(2500)
''')
checks()
git("add", ".", repo=repo)
git("commit", "--quiet", "-m", "Add rice field productivity test", repo=repo)

print("3. Merge into main")
git("switch", "--quiet", "main", repo=repo)
git("merge", "--quiet", "--no-ff", "-m", "Merge branch rice-field-test", "rice-field-test",
    repo=repo)
checks()

print("--- history ---")
print(tidy(git("log", "--graph", "--format=%s", repo=repo)[0]))
```

```
print("Files tracked by git:", len(git("ls-files", repo=repo)[0].splitlines()))
print("Final status (empty = clean):", repr(git("status", "--short", repo=repo)[0]))
```

```
1. Checks on main
   ruff check : All checks passed!
   ruff format: 9 files already formatted
   pytest     : 14 passed
2. New branch: a test for RiceField
   ruff check : All checks passed!
   ruff format: 10 files already formatted
   pytest     : 15 passed
3. Merge into main
   ruff check : All checks passed!
   ruff format: 10 files already formatted
   pytest     : 15 passed
--- history ---
* Merge branch rice-field-test
|\
| * Add rice field productivity test
|/
* Package inventory: classes, tests, and configuration
Files tracked by git: 12
Final status (empty = clean): ''
```

Trace it:

- Step 1: on main, ruff passes, 9 files are tidy, and **14 tests pass**. The first commit.
- Step 2: on the branch `rice-field-test`, one new test file is added. All checks run **before** the commit: 10 files tidy and **15 tests pass**.
- Step 3: merge into `main` with `--no-ff` (always make a merge commit so the branch track shows in history), then the checks again: still 15 tests pass.
- Git tracks **12 files**: 10 package files, one new test file, and `.gitignore`. The final status is empty: nothing left behind.

Exercises

1. Run `01_init_commit.py`, then open the folder `_tmp/basic_repo` in a terminal and type `git log` and `git status`. Compare with the book's output.
2. Change `03_branch_merge.py` so the second branch adds a file `pen.txt`. Does the merge still run without conflict? Why?
3. Add `.idea/` and `*.log` to the `.gitignore` in `05_gitignore_secrets.py`, make a file `debug.log`, and check with `git check-ignore -v debug.log`.
4. **Challenge**: change `04_conflict.py` so the conflict is resolved by **computing the average** of the two numbers (530 g) rather than choosing one. Why can git not make a decision like this by itself?

Chapter Summary

- **git** records history: `git add` chooses, `git commit` saves, `git status` and `git diff` inspect, `git restore` throws away uncommitted changes.
- A **branch** = a side track; **merge** joins it. A **conflict** appears when the same line is changed; a human decides, then commits.
- **.gitignore** keeps temporary files and secrets away. A secret that was committed must be treated as leaked.
- **GitHub** keeps repos on the internet; **issues**, **pull requests**, and **reviews** organise teamwork. This part is only explained, not run.
- **GitHub Actions** runs automatic checks from a YAML file in `.github/workflows/`; the example here only had its syntax checked.
- The daily flow: branch, change, ruff, tests, commit, pull request, review, merge.

Closing Series I1

Congratulations! You have finished **Series I1** and passed the first gate of the Intermediate level. Your ability has shifted from “writing a program that runs” to “building a project that is maintained”:

- Turning field data into **classes**: `Unit` and its children, with inheritance, polymorphism, and `dataclass` (Chapter 16).
- Giving every project its own **environment** with `venv`, `pip`, `uv`, and `pyproject.toml` (Chapter 17).
- Guarding code with **pytest tests** (Chapter 18) and **ruff**, pre-commit, and logging (Chapter 19).
- Keeping history and working together with **git** and the GitHub workflow (Chapter 20).

Your inventory project is now tidy and safe to change. But all its data is still made-up data we typed ourselves. In **Series I2**, your program starts to **fetch and process data from outside**: the internet, large files, and databases.

Next, in Chapter 21: working with APIs, how a program asks a service on the internet for data and reads the reply.

Chapter 21 — Working with APIs: Asking Another Computer for Data

Series I2 — Data and the Internet

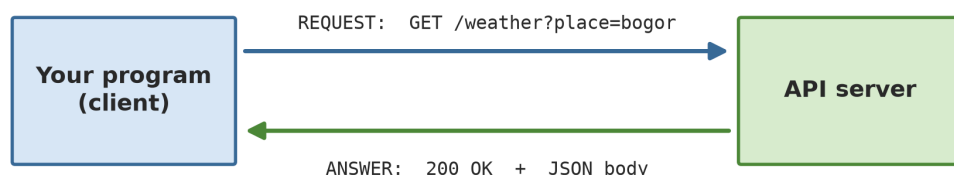
In this chapter, you will:

- Understand what an **API**, **HTTP**, and **JSON** are, using a restaurant waiter as an analogy
- Send requests with `requests` and `httpx`, and read the answers
- Use parameters and a **token** (an entry key) without writing secrets in code
- Handle network errors: status codes, **timeouts**, and retries
- Fetch weather and price data for Running Project I2

All scripts for this chapter are in the `code/ch21/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Series I1 made your code tidy. Now that code needs **ingredients**, and many of them live on other computers: weather forecasts on a weather agency’s server, prices on a market’s server. Picture a restaurant. You (the program) sit at a table, the **waiter** (the API) takes your order, and the **kitchen** (the server) cooks. You never enter the kitchen; you just order in the agreed way and receive a plate with the answer on it. An **API** (*Application Programming Interface*) is that “menu and ordering rules”.

One conversation with an API: request and answer



What the status codes mean



Figure 0.1: One conversation with an API: you send a request, the server answers with a status code and a JSON body.

HTTP and JSON in five minutes

That conversation uses a language called **HTTP**. A **request** contains: a method (`GET` = “please give me”), an address, and sometimes extras such as parameters or a token. An **answer** contains: a **status code** (a three-digit number), *headers* (notes), and a body. The body from an API is usually **JSON**, text whose layout looks like Python dicts and lists (Chapter 9).

Code	Meaning	Your action
200	Success	Read the body

Code	Meaning	Your action
400 / 404	Bad request / address does not exist	Check the address and parameters; repeating will not help
401	A correct token is needed	Check the token
500s (e.g. 503)	Server trouble or busy	You may retry later

A mock server: our own kitchen

To make the examples in this book always give the same output, and to avoid burdening other people's servers, we build a **mock API server** on our own computer. Its helper file `mock_server.py` (in the chapter folder, not an example script) uses Python's built-in `http.server`. It starts at the local address `127.0.0.1` on a random port, serves a few addresses with **fictional** data, and is switched off when the `with` block ends. To `requests`, it looks like a real API.

The first request with requests

`requests` is the best-known third-party library for HTTP (already in this project's environment; on your computer: `pip install requests`).

```
# Chapter 21 - Your first HTTP request with requests
import requests
from mock_server import MockServer

with MockServer() as server:          # starts the mock server on your own computer
    response = requests.get(f"{server.address}/weather?place=bogor&days=2", timeout=5)

    print("Status code :", response.status_code)
    print("Content type:", response.headers["Content-Type"])
    print("Raw text    :", response.text)
# leaving the `with` block: the server is switched off again
```

```
Status code : 200
Content type: application/json; charset=utf-8
Raw text    : {"place": "bogor", "units": {"max_temp": "C", "rain_mm": "mm"}, "daily":
{"date": ["2026-01-01", "2026-01-02"], "max_temp": [30.1, 31.4], "rain_mm": [12.5,
0.0]}}
```

Code 200 means success. `Content-Type` tells us the body is JSON. The “raw text” was sent by the server as one long line; next we turn it into Python data.

JSON and parameters

```
# Chapter 21 - JSON and parameters (params)
import json
import requests
from mock_server import MockServer

with MockServer() as server:
    # params= builds the "?key=value" part correctly (including character encoding)
    response = requests.get(f"{server.address}/weather", params={"place": "bandung",
"days": 3}, timeout=5)
    print("Full address:", response.url.replace(server.address, "<server>"))

    data = response.json()          # JSON -> Python dict/list
    print("Data type    :", type(data).__name__)
    print("Place       :", data["place"])
```

```

print("Dates      :", data["daily"]["date"])
print("Rain (mm)   :", data["daily"]["rain_mm"])
print("Total rain  :", round(sum(data["daily"]["rain_mm"]), 1), "mm")

# the other direction: Python -> JSON text
print(json.dumps({"place": "bandung", "total_rain_mm": 19.2}, ensure_ascii=False,
indent=2))

```

```

Full address: <server>/weather?place=bandung&days=3
Data type    : dict
Place        : bandung
Dates        : ['2026-01-01', '2026-01-02', '2026-01-03']
Rain (mm)    : [5.0, 0.0, 14.2]
Total rain   : 19.2 mm
{
  "place": "bandung",
  "total_rain_mm": 19.2
}

```

- `params={...}` builds the `?place=bandung&days=3` part correctly. Do not assemble addresses by hand.
- `.json()` turns the answer into a dict/list. Three days of Bandung rain: $5.0 + 0.0 + 14.2 = \mathbf{19.2\text{ mm}}$.
- `json.dumps(...)` is the reverse direction: from Python to JSON text (`indent=2` tidies it up).

httpx: the sibling of requests

`httpx` is similar to `requests` but has a **Client**: an object that remembers the base address and settings so they are not repeated in every call (it also supports an asynchronous mode, covered in later series).

```

# Chapter 21 - httpx: a sibling of requests, with a reusable Client
import httpx
from mock_server import MockServer

with MockServer() as server:
    # A Client remembers the base address and settings, so you do not repeat them in
    every call
    with httpx.Client(base_url=server.address, timeout=5.0) as client:
        for place in ["bogor", "bandung", "malang"]:
            response = client.get("/weather", params={"place": place, "days": 5})
            data = response.json()
            rain = data["daily"]["rain_mm"]
            print(f"{place:8} status {response.status_code} total rain {sum(rain):5.1f}
mm")

```

```

bogor    status 200  total rain  49.2 mm
bandung  status 200  total rain  51.5 mm
malang   status 200  total rain  28.7 mm

```

Three places, five-day rain totals of 49.2, 51.5, and 28.7 mm (fictional data). As a beginner, pick one of the two first; the knowledge is nearly the same.

Authentication: showing your “membership card”

Many APIs serve only holders of a **token** (a kind of membership card). The most common way: send the token in the header **Authorization: Bearer <token>**. Other ways: an **API key** in a header or parameter, or **Basic auth** (a user name and password; in `requests` via `auth=(name, password)`).

```

# Chapter 21 - Basic authentication: a token in the Authorization header
import os
import requests

```



```

from mock_server import MockServer

with MockServer() as server:
    address = f"{server.address}/secret"

    without = requests.get(address, timeout=5)
    print("Without token:", without.status_code, without.json())

    # In the real world the token is read from an environment variable, NEVER written
    in code.
    # Here we park the random token of our mock server in a temporary environment variable.
    os.environ["SAMPLE_API_TOKEN"] = server.token
    token = os.environ["SAMPLE_API_TOKEN"]

    with_token = requests.get(address, headers={"Authorization": f"Bearer {token}"},
    timeout=5)
    print("With token    :", with_token.status_code, with_token.json())

```

```

Without token: 401 {'error': 'token missing or wrong'}
With token    : 200 {'message': 'you are in', 'officer': 'fictional'}

```

Without a token the server answers 401; with one it answers 200. Notice how the token is treated: read from an **environment variable** (`os.environ`), not written in the code. Here the token is random and made by the mock server, so no real secret is involved.

⚠ WARNING

Never write a token, API key, or password straight into code, least of all code that will be uploaded to GitHub (Chapter 20). Keep it in an environment variable or in an `.env` file listed in `.gitignore`.

When things do not go smoothly

Networks are unpredictable: wrong addresses, dead servers, dropped connections. A good program **expects** this.

Status codes and `raise_for_status()`

```

# Chapter 21 - Error status codes and raise_for_status()
import requests
from mock_server import MockServer

with MockServer() as server:
    for address in ["/weather?place=bogor&days=1", "/weather?place=atlantis", "/weather", "/secret"]:
        response = requests.get(server.address + address, timeout=5)
        print(f"{address:28} -> {response.status_code}")

    # raise_for_status() turns a 4xx/5xx code into an exception you can catch
    try:
        requests.get(f"{server.address}/weather?place=atlantis",
        timeout=5).raise_for_status()
    except requests.HTTPError as error:
        print("Caught HTTPError:", error.response.status_code, error.response.json())

```

```

/weather?place=bogor&days=1  -> 200
/weather?place=atlantis      -> 404
/weather                     -> 400

```

```
/secret -> 401
Caught HTTPError: 404 {'error': 'unknown place'}
```

requests does **not** treat 404 as an error; it is merely an answer with a different code. `raise_for_status()` turns 4xx/5xx codes into an `HTTPError` exception that you can `try/except` (Chapter 12).

Timeout: do not wait forever

```
# Chapter 21 - Timeout: do not wait forever
import requests
from mock_server import MockServer

with MockServer() as server:          # the /slow address deliberately answers after
1.5 seconds
    try:
        requests.get(f"{server.address}/slow", timeout=0.3)
    except requests.Timeout:
        print("The server took too long (> 0.3 seconds): request cancelled.")

        response = requests.get(f"{server.address}/slow", timeout=5)
        print("With a 5 second limit:", response.json()["message"])

# Without timeout=, requests waits without any time limit (see the official
documentation).
```

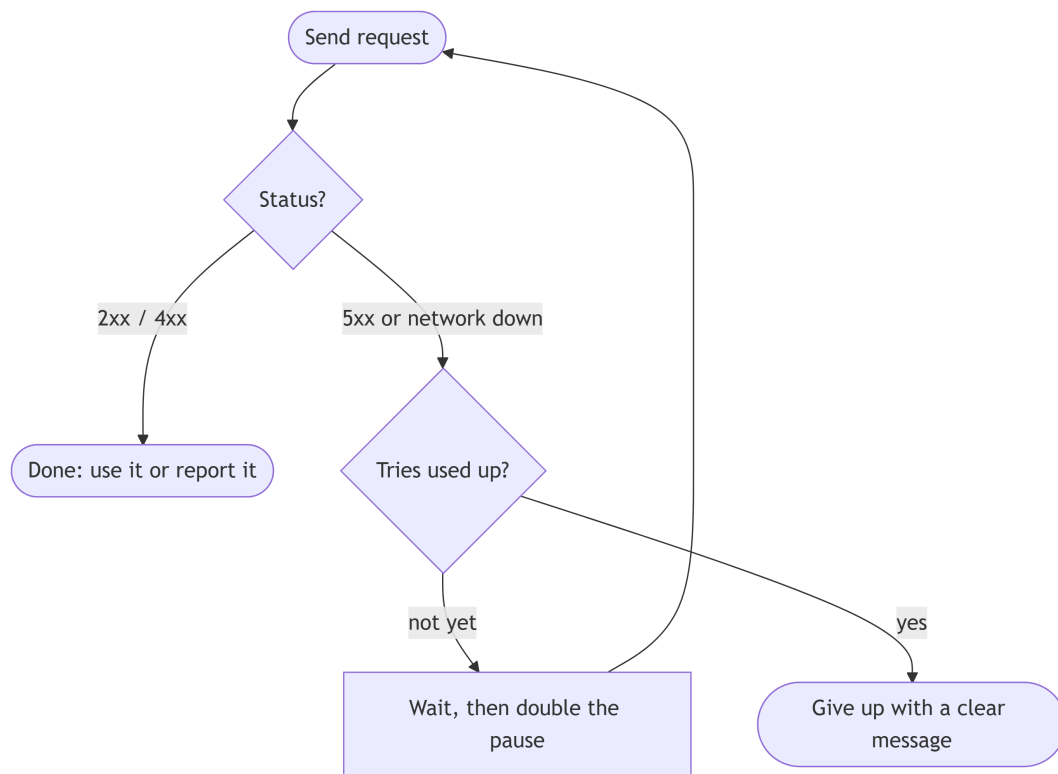
```
The server took too long (> 0.3 seconds): request cancelled.
With a 5 second limit: finally done
```

i REMEMBER

According to the official `requests` documentation, if `timeout=` is not given, the request has **no time limit** and your program can hang. Make a habit of always setting it. In `httpx`, the default limit is 5 seconds.

Retry with a pause

5xx errors are often temporary. The strategy: try again with a growing pause (*exponential backoff*), but only for errors worth repeating, and stop after a few tries.



The retry flow: only temporary errors (5xx, network) are repeated, with a doubling pause.

```

# Chapter 21 - Retry with a growing pause
import time
import requests
from mock_server import MockServer

def fetch_with_retry(address, max_tries=4, first_pause=0.05):
    pause = first_pause
    for attempt in range(1, max_tries + 1):
        try:
            response = requests.get(address, timeout=5)
            if response.status_code < 500:      # 2xx/4xx: repeating is pointless
                return response
            print(f"  attempt {attempt}: status {response.status_code}, wait
{pause:.2f} seconds")
        except (requests.ConnectionError, requests.Timeout):
            print(f"  attempt {attempt}: network trouble, wait {pause:.2f} seconds")
            time.sleep(pause)
            pause = pause * 2                  # doubling pause: 0.05 -> 0.10 -
> 0.20 ...
            raise RuntimeError("giving up after several tries")

with MockServer() as server:                # /busy fails twice (503), then succeeds
    result = fetch_with_retry(f"{server.address}/busy")
    print("Success:", result.json())
  
```

```

attempt 1: status 503, wait 0.05 seconds
attempt 2: status 503, wait 0.10 seconds
Success: {'message': 'success', 'attempt': 3}
  
```

The mock server deliberately fails twice (503), so the 3rd attempt succeeds. The pause doubles each time: 0.05 then 0.10 seconds (in the real world start at 1 second or more so you do not hammer the server).

One call to a real API: Open-Meteo

Open-Meteo offers weather forecasts through an open API. What I found when checking its terms on 2026-10-10 (open-meteo.com/en/terms):

- Free use is for **non-commercial** purposes only and needs **no key**; the limit is under 10,000 calls per day, 5,000 per hour, and 600 per minute.
- Using the free API means accepting the **CC BY 4.0** licence, which **requires attribution**. That is why the script prints “Weather data by Open-Meteo.com”.
- This book is not a commercial application; readers building paid products must read those terms themselves.

```
# Chapter 21 - One call to a real API: Open-Meteo (no key)
import requests

address = "https://api.open-meteo.com/v1/forecast"
parameters = {
    "latitude": -6.6,                # around Bogor
    "longitude": 106.8,
    "daily": "temperature_2m_max,precipitation_sum",
    "timezone": "Asia/Jakarta",
    "forecast_days": 3,
}
response = requests.get(address, params=parameters, timeout=10)
response.raise_for_status()
data = response.json()

print("Status      :", response.status_code)
print("Units       :", data["daily_units"])
for day, temp, rain in zip(data["daily"]["time"], data["daily"]["temperature_2m_max"],
                           data["daily"]["precipitation_sum"]):
    print(f"{day} max temp {temp} C rain {rain} mm")
print("Weather data by Open-Meteo.com (CC BY 4.0 licence)")
```

```
Status      : 200
Units       : {'time': 'iso8601', 'temperature_2m_max': '°C', 'precipitation_sum':
'mm'}
2026-10-10 max temp 33.8 C rain 1.0 mm
2026-10-11 max temp 32.9 C rain 5.2 mm
2026-10-12 max temp 32.4 C rain 6.6 mm
Weather data by Open-Meteo.com (CC BY 4.0 licence)
```

This code was **not re-run** when the book was built because it needs the internet and its result changes every day. The output above was **taken by me on 2026-10-10** and stored. When you run it, the dates and numbers will be **different**; that is normal.

TIP

About **commodity prices**: on 2026-10-10 I could not find a verified agricultural commodity price API that is free and needs no key. The list of 71 World Bank API data sources has no commodity price source, and the commodity price services that showed up in a web search mostly ask for registration or a key (I did not test their terms). So the prices in this book come from the mock server (fictional). For a real project, check official sources such as a ministry or a commodity exchange, and read their terms of use.

Running Project I2: fetching weather and price data

Step 1 of the Series I2 project: fetch the weather of three places and the prices of four commodities from our “API”, using a small function that stops clearly on any error, then save them as raw JSON in the `_tmp` folder.

```
# Chapter 21 - Running project I2, step 1: fetching weather and price data
import json
from pathlib import Path

import requests
from mock_server import MockServer

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

def fetch_json(address, **param):
    """Fetches JSON; stops with a clear message on any error."""
    response = requests.get(address, params=param, timeout=5)
    response.raise_for_status()
    return response.json()

with MockServer() as server:
    weather = {}
    for place in ["bogor", "bandung", "malang"]:
        weather[place] = fetch_json(f"{server.address}/weather", place=place)["daily"]
    prices = fetch_json(f"{server.address}/prices")["data"]

(tmp / "raw_weather.json").write_text(json.dumps(weather, ensure_ascii=False,
    indent=1), encoding="utf-8")
(tmp / "raw_prices.json").write_text(json.dumps(prices, ensure_ascii=False, indent=1),
    encoding="utf-8")

print("Weather places :", list(weather))
print("Days per place :", len(weather["bogor"]["date"]))
print("Price rows      :", len(prices), "| example:", prices[0])
print("Saved in the _tmp folder: raw_weather.json, raw_prices.json")

Weather places : ['bogor', 'bandung', 'malang']
Days per place : 5
Price rows      : 20 | example: {'date': '2026-01-01', 'commodity': 'rice',
'price_per_kg': 6200}
Saved in the _tmp folder: raw_weather.json, raw_prices.json
```

Weather for three places, five days each, and 20 price rows (4 commodities x 5 days). This raw data will be cleaned (Chapter 22), merged (Chapter 23), stored (Chapter 24), and drawn (Chapter 25). So that every script stands on its own, the next chapters copy the same data directly into their code.

Exercises

1. Change `02_json_and_params.py` to fetch `malang` and print the day with the most rain.
2. In `05_error_status.py`, add the address `/no-such-page`. Which status code appears?
3. Change the `timeout` in `06_timeout.py` to 2 seconds. What changes, and why?
4. **Challenge:** write a function `fetch_safely(address)` that retries on 5xx but **stops immediately** (no repeat) on 404, and returns `None` if it finally fails.

Chapter Summary

- **API** = the official way to ask another program for data; **HTTP** = its language; **JSON** = the shape of the content.
- `requests.get(address, params=..., timeout=...)`; `.json()` turns the answer into Python data. `httpx.Client` is similar, with shared settings.
- Status codes: 2xx success, 4xx our fault, 5xx the server's fault. `raise_for_status()` turns errors into exceptions.
- Always use a **timeout**; retry only temporary errors, with a doubling pause.
- Tokens are read from environment variables, **not** written in code.
- Open-Meteo: free for non-commercial use, no key, attribution required (CC BY 4.0). A free, keyless commodity price API: not found (unverified).

Next, in Chapter 22: incoming data is not always tidy, for example field notes typed by humans. We learn **regex** to dig out and clean text.

Chapter 22 — Regex and Text Processing: Digging Data out of Messy Writing

Series I2 — Data and the Internet

In this chapter, you will:

- Understand **regex** (*regular expression*): a tiny language that describes text patterns
- Find and extract pieces of text with `re.search`, `re.findall`, and `groups`
- Replace and split text with `re.sub` and `re.split`
- Recognise classic traps: greedy repeaters and forgetting to check for `None`
- Clean fictional field notes for Running Project I2

All scripts for this chapter are in the `code/ch22/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Data from an API (Chapter 21) is usually tidy. Data from humans is not. One field officer types “dia 32,5 cm”, a colleague “diameter=32.5cm”, another “TEAK” in all capitals. Filtering writing like that with `if` and `split` quickly turns into a jungle. **Regex** is like a **stencil card for patterns**: you punch holes in the card in the shape you are looking for (“two digits, a slash, two digits...”), lay it over the text, and whatever fits the holes is what you take out.

Regex cuts one note line into named parts

```
pattern
(?P<day>\d{2})/(?P<mon>\d{2})/(?P<year>\d{4}) \|| Plot (?P<plot>\w) \|| (?P<kind>\w+) \|| dia (?P<dia>[\d,]+) cm

10 / 01 / 2026 | Plot A | Teak | dia 32,5 cm
day mon year plot kind dia

m.groupdict()
{'day': '10', 'mon': '01', 'year': '2026', 'plot': 'A', 'kind': 'Teak', 'dia': '32,5'}
```

Figure 0.1: One pattern with named groups cuts a note line into parts that are easy to use.

Searching: search, match, findall

The `re` module comes with Python; nothing to install.

```
# Chapter 22 - Basics of the re module: search, match, fullmatch, findall
import re

text = "Plot A: diameter 32,5 cm, height 18 m"

print(re.search(r"diameter", text))          # looks ANYWHERE
print(re.match(r"diameter", text))           # match: only from the START of
the text
print(re.match(r"Plot", text) is not None)
print(re.fullmatch(r"\d+", "2026"), re.fullmatch(r"\d+", "2026 "))
```



```

found = re.search(r"\d+,\d+", text)           # a decimal number with a comma
print(found.group(), "is at position", found.start(), "to", found.end())

print(re.findall(r"\d+", text))                # all runs of digits
for hit in re.finditer(r"[A-Z]", text):        # every capital letter with
its position                                  # its position
    print(hit.group(), hit.start(), end=" | ")
print()

```

```

<re.Match object; span=(8, 16), match='diameter'>
None
True
<re.Match object; span=(0, 4), match='2026'> None
32,5 is at position 17 to 21
['32', '5', '18']
P 0 | A 5 |

```

- `re.search` looks **anywhere**; the result is a *Match* object (with a position such as `span=(8, 16)`) or `None` if nothing is found.
- `re.match` only succeeds if the pattern is at the **start** of the text, which is why `diameter` gives `None`.
- `re.fullmatch` demands that the **whole** text matches: `"2026"` passes, `"2026 "` (with a space) fails.
- `findall` gives a list of all matches: three runs of digits `['32', '5', '18']`. Notice: `32,5` is split into `32` and `5`, because the pattern `\d+` knows nothing about commas.
- The `r"..."` prefix (*raw string*) makes backslashes be read as they are. Always use it for patterns.

Ingredients of patterns

Pattern	Meaning
<code>\d</code>	one digit; <code>\w</code> one letter/digit/underscore; <code>\s</code> one space (or tab)
<code>.</code>	any one character (except a newline)
<code>[A-C]</code>	one character from that set; <code>[^]</code> = anything except a quote mark
<code>+ / * / ?</code>	one or more / zero or more / zero or one
<code>{4} / {2,3}</code>	exactly 4 times / 2 to 3 times
<code>(...)</code>	group; <code>(?:...)</code> unnumbered group; <code>(?P<name>...)</code> named group
<code>\b</code>	or; <code>\b</code> word boundary

```

# Chapter 22 - Common patterns: character classes and repeaters
import re

text = "Plot A: 32,5 cm; Plot B: 45 cm; code JT-0042; Plot C: 28,0 cm"

patterns = {
    r"\d": "one digit",
    r"\d+": "one or more digits",
    r"\d+,\d+": "a number with a comma",
    r"\d+(?:,\d+)?": "a number, the comma part optional",
    r"[A-C]": "one letter between A and C",
    r"Plot \w": "the word Plot, a space, one letter/digit",
    r"[A-Z]{2}-\d{4}": "two capital letters, a minus, four digits",
}

```

```

}
for p, meaning in patterns.items():
    print(f"{p:20} {meaning:44} -> {re.findall(p, text)}")

```

<code>\d</code>	one digit	-> ['3', '2', '5', '4', '5', '0', '0', '4', '2', '2', '8', '0']
<code>\d+</code>	one or more digits	-> ['32', '5', '45', '0042', '28', '0']
<code>\d+,\d+</code>	a number with a comma	-> ['32,5', '28,0']
<code>\d+(?:,\d+)?</code>	a number, the comma part optional	-> ['32,5', '45', '0042', '28,0']
<code>[A-C]</code>	one letter between A and C	-> ['A', 'B', 'C']
<code>Plot \w</code>	the word Plot, a space, one letter/digit	-> ['Plot A', 'Plot B', 'Plot C']
<code>[A-Z]{2}-\d{4}</code>	two capital letters, a minus, four digits	-> ['JT-0042']

Trace the lines: `\d+(?:,\d+)?` captures 32,5 and 45 together because the comma part is made *optional* (?). The last pattern finds the code JT-0042.

TIP

Do not write a long pattern all at once. Write one piece, test it with `findall` on sample text, then add the next piece. Sites such as regex101.com help you see matches visually.

Groups: taking out the parts

Brackets make a **group**, a part of the pattern whose result can be taken separately. Named groups (`?P<name>`) are far easier to read than numbers.

```

# Chapter 22 - Groups: picking out the matched parts
import re

line = "10/01/2026 | Plot A | Teak | dia 32,5 cm"

# plain brackets = numbered group
m = re.search(r"(\d{2})/(\d{2})/(\d{4})", line)
print(m.groups())                    # ('10', '01', '2026')
print("Year:", m.group(3))

# (?P<name>...) = named group: easier to read
pattern = r"(?P<day>\d{2})/(?P<mon>\d{2})/(?P<year>\d{4}) \| Plot (?P<plot>\w) \| (?P<kind>\w+)"
m = re.match(pattern, line)
print(m.groupdict())
print(f"{m['year']}-{m['mon']}-{m['day']}")    # building an ISO date

```

```

('10', '01', '2026')
Year: 2026
{'day': '10', 'mon': '01', 'year': '2026', 'plot': 'A', 'kind': 'Teak'}
2026-01-10

```

`m.groups()` gives a tuple, `m.groupdict()` gives a dict, and `m['year']` takes one group. We rebuild the date 10/01/2026 as 2026-01-10 (ISO format, easy to sort).

Replacing and splitting

```
# Chapter 22 - Replacing (sub) and splitting (split)
import re

text = "dia 32,5 cm; dia 45,2 cm; dia 28 cm"

print(re.sub(r"(\d),(\d)", r"\1.\2", text))          # decimal comma -> dot; \1 \2 =
groups 1 and 2
print(re.sub(r"\s+", " ", "Plot   A   Teak"))      # many spaces -> one space
print(re.subn(r"cm", "CM", text))                  # subn also gives the number
of replacements

def to_metres(hit):
    """Called for every match; returns the replacement text."""
    cm = float(hit.group(1).replace(",", "."))
    return f"{cm / 100:.3f} m"

print(re.sub(r"(\d+(?:,\d+)?) cm", to_metres, text))
print(re.split(r"\s*[;,:]\s*", "A ; B,C |D"))      # split at any one of the separators
```

```
dia 32.5 cm; dia 45.2 cm; dia 28 cm
Plot A Teak
('dia 32,5 CM; dia 45,2 CM; dia 28 CM', 3)
dia 0.325 m; dia 0.452 m; dia 0.280 m
['A', 'B', 'C', 'D']
```

- `re.sub(pattern, replacement, text)` replaces all matches. `\1` and `\2` in the replacement refer to groups 1 and 2, so `32,5` becomes `32.5`.
- `re.subn` also reports the number of replacements: `3`.
- The replacement may be a **function** called for each match: we turn centimetres into metres (`32,5 cm` becomes `0.325 m`).
- `re.split` cuts at any one of the separators, even with spaces around them.

compile, flags, and VERBOSE

```
# Chapter 22 - compile, flags (IGNORECASE) and VERBOSE patterns
import re

# A pattern used repeatedly: compile once, use many times
species_pattern = re.compile(r"\b(teak|meranti|mahogany|sengon)\b", re.IGNORECASE)
print(species_pattern.findall("TEAK healthy, Meranti sick, mahogany, another tree"))

# VERBOSE: spaces and comments inside the pattern are ignored, so long patterns can
be explained
date_pattern = re.compile(r"""
    (?P<day>\d{1,2})      # day
    [/-]                  # separator: slash or minus
    (?P<mon>\d{1,2})      # month
    [/-]
    (?P<year>\d{4})        # year
    """, re.VERBOSE)
for s in ["10/01/2026", "5-3-2026", "2026-01-10"]:
    m = date_pattern.fullmatch(s)
    print(f"{s:11} ->", m.groupdict() if m else "no match")
```

```
['TEAK', 'Meranti', 'mahogany']
10/01/2026 -> {'day': '10', 'mon': '01', 'year': '2026'}
5-3-2026 -> {'day': '5', 'mon': '3', 'year': '2026'}
2026-01-10 -> no match
```

`re.compile` prepares a pattern once for repeated use. `re.IGNORECASE` ignores upper/lower case (which is why `TEAK` is included). `re.VERBOSE` allows spaces and comments inside the pattern, so a long pattern can be explained line by line. Notice that `2026-01-10` does **not match** because our pattern expects the year at the end.

Traps: greedy and forgetting None

```
# Chapter 22 - Trap: greedy and lazy repeaters
import re

text = 'note "healthy" then note "sick" finished'

print(re.findall(r'".*"', text))      # greedy: swallows as much as possible
print(re.findall(r'".*?"', text))    # lazy (?): stops as soon as possible
print(re.findall(r'("[^"]*)"', text)) # another way: "anything except a quote mark"
```

```
['"healthy" then note "sick"']
['"healthy"', '"sick"']
['healthy', 'sick']
```

The pattern `".*"` is **greedy**: it swallows as much as it can, from the first quote to the last quote, so two notes merge into one. Adding `?` makes it **lazy**: it stops at the nearest quote. The third way, `[^"]*`, is often the clearest.

⚠ WARNING

`re.search` returns `None` when nothing matches. Calling `.group()` on `None` raises an `AttributeError`. Always check first: `if result:`.

```
# Chapter 22 - Two classic errors: a broken pattern and forgetting to check for None
import re

try:
    re.compile(r"(\d+")                # opening bracket without its partner
except re.error as error:
    print("Broken pattern:", error)

result = re.search(r"\d+", "there is no number here")
print("Search result:", result)        # None, not an error

try:
    print(result.group())
except AttributeError as error:
    print("Error:", error)

# The safe way: check first
if result:
    print(result.group())
else:
    print("No number; carrying on safely.")
```

```
Broken pattern: missing ), unterminated subpattern at position 0
Search result: None
```

```
Error: 'NoneType' object has no attribute 'group'
No number; carrying on safely.
```

A pattern with an unpaired bracket raises `re.error` at `compile` time. The wording of the error message can differ between Python versions; what matters is that you know the error lives in the **pattern**, not in the text.

Running Project I2: cleaning field notes

Step 2: five fictional field notes written in different styles, plus one broken line. One `VERBOSE` pattern accepts all the styles, then the code tidies up: dates become ISO, plots become capital letters, species get an initial capital, comma numbers become Python decimals, and “healthy/sick” becomes `True/False`.

```
# Chapter 22 - Running project I2, step 2: cleaning field notes with regex
import json
import re
from pathlib import Path

notes = [
    "10/01/2026 | Plot A | Teak | dia 32,5 cm | height 18 m | HEALTHY",
    "10/01/2026 | plot a | meranti | diameter=45.2cm | height: 25,5 m | sick",
    "11-01-2026 | Plot B | Mahogany | dia 28 cm | height 15 m | healthy",
    "11/01/2026 | Plot C | sengon | dia 19,8cm | height 12 m | Healthy ",
    "12/01/2026 | Plot B | TEAK | dia 36,1 cm | height 20,5 m | sick",
    "broken: a line without clear data",
]

PATTERN = re.compile(r"""
    (?P<day>\d{1,2})[/-](?P<mon>\d{1,2})[/-](?P<year>\d{4}) \s*\| \s*
    plot\s+(?P<plot>[a-z]) \s*\| \s*
    (?P<species>[a-z]+) \s*\| \s*
    (?:dia|diameter)\s*[:=]\s*(?P<dia>\d+(?:[.],\d+)?)\s*cm \s*\| \s*
    height\s*[:=]\s*(?P<height>\d+(?:[.],\d+)?)\s*m \s*\| \s*
    (?P<condition>healthy|sick)
""", re.VERBOSE | re.IGNORECASE)

def number(text):
    return float(text.replace(",", "."))

clean, rejected = [], []
for line in notes:
    m = PATTERN.fullmatch(line.strip())
    if m is None:
        rejected.append(line)
        continue
    clean.append({
        "date": f"{m['year']}-{int(m['mon']):02d}-{int(m['day']):02d}",
        "plot": m["plot"].upper(),
        "species": m["species"].capitalize(),
        "diameter_cm": number(m["dia"]),
        "height_m": number(m["height"]),
        "healthy": m["condition"].lower() == "healthy",
    })

for c in clean:
    print(c)
print("Rejected:", rejected)
```

```
tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
(tmp / "clean_notes.json").write_text(json.dumps(clean, ensure_ascii=False, indent=1),
encoding="utf-8")
print(f"{len(clean)} clean rows, {len(rejected)} rejected")
```

```
{'date': '2026-01-10', 'plot': 'A', 'species': 'Teak', 'diameter_cm': 32.5, 'height_m':
18.0, 'healthy': True}
{'date': '2026-01-10', 'plot': 'A', 'species': 'Meranti', 'diameter_cm': 45.2,
'height_m': 25.5, 'healthy': False}
{'date': '2026-01-11', 'plot': 'B', 'species': 'Mahogany', 'diameter_cm': 28.0,
'height_m': 15.0, 'healthy': True}
{'date': '2026-01-11', 'plot': 'C', 'species': 'Sengon', 'diameter_cm': 19.8,
'height_m': 12.0, 'healthy': True}
{'date': '2026-01-12', 'plot': 'B', 'species': 'Teak', 'diameter_cm': 36.1, 'height_m':
20.5, 'healthy': False}
Rejected: ['broken: a line without clear data']
5 clean rows, 1 rejected
```

Of six lines, **5 are clean and 1 is rejected**. Rejecting honestly is better than guessing: rejected lines are recorded so a human can check them. The clean result is saved in `_tmp/clean_notes.json`, ready to be merged with other data in the next chapter.

Exercises

1. Write a pattern that finds every code like JT-0042 in "JT-0042, MR-0107, xx-12". (Only two results.)
2. With `re.sub`, turn "Plot A, Plot B , Plot C" into "Plot A, Plot B, Plot C".
3. Add a new note line to `08_project_clean.py` in a style not yet covered (e.g. height=14 m without spaces). Is it accepted? Change the pattern if needed.
4. **Challenge:** extend the pattern so that a temperature such as `temp 27,5 C` at the end of a line becomes an optional group (may be present or not).

Chapter Summary

- **Regex** = a language for text patterns; the `re` module is built into Python; write patterns as `r"..."`.
- `search` (anywhere), `match` (from the start), `fullmatch` (the whole text), `findall` (all).
- **Groups** (...) and `(?P<name>...)` take out parts; `groupdict()` gives a dict.
- `sub` replaces (optionally with a function), `split` splits, `compile` prepares a pattern, `VERBOSE` makes patterns readable.
- Watch out for **greedy** repeaters and always check for a `None` result.
- Dirty data is cleaned with one clear pattern; lines that do not match are **rejected and recorded**.

Next, in Chapter 23: the data is clean, but tens of thousands of rows do not fit ordinary loops. Meet **numpy** and **pandas**.

Chapter 23 — Introduction to numpy and pandas: Smart Tables for Field Data

Series I2 — Data and the Internet

In this chapter, you will:

- Use **numpy arrays**: one formula for thousands of numbers at once
- Create and read pandas **DataFrames**, tables with column names
- Read and write CSV, Excel, and JSON
- Filter rows and summarise data with `groupby`
- Merge five sectors, weather, and prices: Running Project I2

All scripts for this chapter are in the `code/ch23/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Imagine a field notebook with thousands of rows. Calculating the mean diameter with `for` is possible, but slow and long. **numpy** is like a **uniform sack**: everything inside is one kind (say, decimal numbers), so it can be weighed, multiplied, and summed all at once. **pandas** is a **labelled worksheet** (like Excel) built on numpy: every column has a name, every row has a number, and you can filter and summarise it in one or two lines of code. (Installing: `pip install numpy pandas`; both are already in this project's environment.)

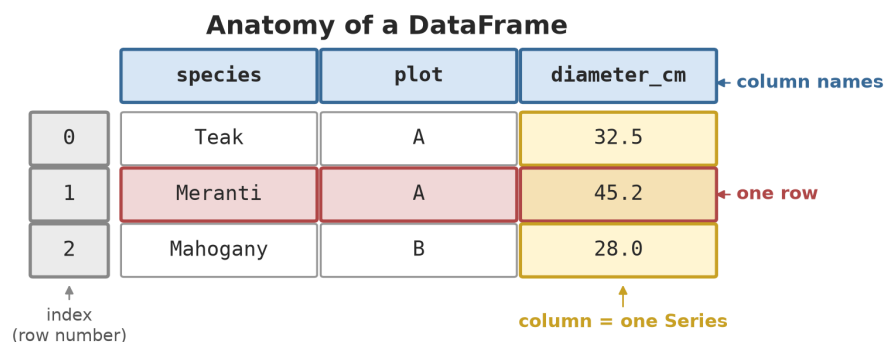


Figure 0.1: Anatomy of a DataFrame: the index on the left, column names on top, one column is one Series, one row is one record.

numpy: arrays

```
# Chapter 23 - numpy arrays: fast, uniform lists of numbers
import numpy as np

diameter = np.array([32.5, 45.2, 28.0, 19.8, 36.1, 51.5])
print(diameter)
print("type:", diameter.dtype, "| shape:", diameter.shape, "| count:", diameter.size)

print(np.arange(0, 10, 2))          # like range(), but produces an array
```



```
print(np.linspace(0, 1, 5))          # 5 evenly spaced numbers between 0 and 1
print(np.zeros((2, 3)))              # a table of 2 rows x 3 columns filled
with zeros
```

```
table = np.array([[32.5, 18.0], [45.2, 25.5], [28.0, 15.0]]) # diameter, height
print(table.shape, "->", table.shape[0], "rows,", table.shape[1], "columns")
print(table[1, 0], "|", table[:, 1], "|", table[0])          # [row, column]
```

```
[32.5 45.2 28.  19.8 36.1 51.5]
type: float64 | shape: (6,) | count: 6
[0 2 4 6 8]
[0.  0.25 0.5  0.75 1.  ]
[[0. 0. 0.]
 [0. 0. 0.]]
(3, 2) -> 3 rows, 2 columns
45.2 | [18.  25.5 15. ] | [32.5 18. ]
```

- `np.array(...)` makes an array. `dtype: float64` means everything inside is a 64-bit decimal; `shape: (6,)` means one dimension, six elements.
- `np.arange` is like `range` (Chapter 8); `np.linspace(0, 1, 5)` divides 0 to 1 evenly into 5 points; `np.zeros((2, 3))` makes a table of zeros with 2 rows x 3 columns.
- In a two-dimensional array you write `[row, column]`: `table[1, 0] = 45.2` (row 1, column 0), `table[:, 1] =` the whole of column 1 (height).

Whole-array operations

```
# Chapter 23 - Whole-array operations (vectors), logical selection, and axes
import numpy as np

diameter_cm = np.array([32.5, 45.2, 28.0, 19.8, 36.1, 51.5])

radius_m = diameter_cm / 2 / 100          # one formula for ALL elements, no loop
print("radius (m):", radius_m)
area_m2 = 3.14159 * radius_m ** 2
print("cross-section area (m2):", area_m2.round(3))

big = diameter_cm >= 30                  # the result is an array of True/False
print("big? ", big)
print("the big ones:", diameter_cm[big]) # selecting with a True/False mask
print("mean:", diameter_cm.mean().round(2), "| largest:", diameter_cm.max(), "|
standard deviation:",
      diameter_cm.std().round(2))

table = np.array([[32.5, 18.0], [45.2, 25.5], [28.0, 15.0]])
print("mean of each column (axis=0):", table.mean(axis=0))
print("sum of each row      (axis=1):", table.sum(axis=1))
```

```
radius (m): [0.1625 0.226  0.14  0.099 0.1805 0.2575]
cross-section area (m2): [0.083 0.16  0.062 0.031 0.102 0.208]
big?       [ True  True False False  True  True]
the big ones: [32.5 45.2 36.1 51.5]
mean: 35.52 | largest: 51.5 | standard deviation: 10.51
mean of each column (axis=0): [35.23333333 19.5      ]
sum of each row      (axis=1): [50.5 70.7 43. ]
```

There is no `for`: `diameter_cm / 2 / 100` is carried out on all six elements at once (**vectorisation**). The comparison `diameter_cm >= 30` produces an array of True/False, and that array can be used as a **mask** to select elements (`diameter_cm[big]`).

`axis` sets the direction: `axis=0` summarises **downward** per column (mean diameter 35.23 and height 19.5), `axis=1` summarises **sideways** per row (50.5; 70.7; 43.0).

⚠ WARNING

The numpy standard deviation (`std()`) is **10.51**, yet pandas `describe()` in the DataFrame section below shows **11.51** for the same data. This is not a miscalculation: numpy divides by n (the population) by default, pandas divides by $n-1$ (a sample). Series I3 explains why; for now, remember that two libraries can have different defaults.

pandas: DataFrame

```
# Chapter 23 - DataFrame: a table with column names
import pandas as pd

trees = pd.DataFrame({
    "species": ["Teak", "Meranti", "Mahogany", "Sengon", "Teak", "Meranti"],
    "plot": ["A", "A", "B", "C", "B", "C"],
    "diameter_cm": [32.5, 45.2, 28.0, 19.8, 36.1, 51.5],
    "height_m": [18.0, 25.5, 15.0, 12.0, 20.5, 28.0],
    "healthy": [True, False, True, True, False, True],
})
print(trees)
print()
print("shape (rows, columns):", trees.shape)
print(trees.dtypes)
print()
print(trees["diameter_cm"].mean())      # one column = a Series
print(trees.describe().round(2))      # statistical summary of the numeric columns
```

	species	plot	diameter_cm	height_m	healthy
0	Teak	A	32.5	18.0	True
1	Meranti	A	45.2	25.5	False
2	Mahogany	B	28.0	15.0	True
3	Sengon	C	19.8	12.0	True
4	Teak	B	36.1	20.5	False
5	Meranti	C	51.5	28.0	True

```
shape (rows, columns): (6, 5)
species              str
plot                 str
diameter_cm         float64
height_m            float64
healthy             bool
dtype: object
```

```
35.516666666666666
diameter_cm  height_m
count         6.00      6.00
mean         35.52     19.83
std          11.51      6.12
min          19.80     12.00
25%          29.12     15.75
50%          34.30     19.25
75%          42.92     24.25
max          51.50     28.00
```

The columns of a DataFrame may be of different kinds: text (`str`), decimal (`float64`), true/false (`bool`). A single column, such as `trees["diameter_cm"]`, is called a **Series**. `describe()` summarises the numeric columns: the count, mean (35.52), smallest (19.8), quartiles, largest (51.5).

Reading and writing files

```
# Chapter 23 - Writing and reading CSV, Excel, and JSON
from pathlib import Path
import pandas as pd

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

trees = pd.DataFrame({
    "species": ["Teak", "Meranti", "Mahogany"],
    "diameter_cm": [32.5, 45.2, 28.0],
    "healthy": [True, False, True],
})

trees.to_csv(tmp / "trees.csv", index=False)          # index=False: do not write
row numbers
print((tmp / "trees.csv").read_text(encoding="utf-8"))

from_csv = pd.read_csv(tmp / "trees.csv")
print(from_csv.equals(trees), "<- CSV came back identical")

trees.to_excel(tmp / "trees.xlsx", index=False)        # uses the openpyxl library
from_excel = pd.read_excel(tmp / "trees.xlsx")
print(from_excel.equals(trees), "<- Excel came back identical")

trees.to_json(tmp / "trees.json", orient="records", force_ascii=False)
print((tmp / "trees.json").read_text(encoding="utf-8"))
```

```
species,diameter_cm,healthy
Teak,32.5,True
Meranti,45.2,False
Mahogany,28.0,True

True <- CSV came back identical
True <- Excel came back identical
[{"species":"Teak","diameter_cm":32.5,"healthy":true},
{"species":"Meranti","diameter_cm":45.2,"healthy":false},
{"species":"Mahogany","diameter_cm":28.0,"healthy":true}]
```

`to_csv/read_csv`, `to_excel/read_excel` (needs `openpyxl`), and `to_json/read_json` make exchanging data with other programs as easy as one line. `index=False` prevents the row numbers from being written as a column. The `equals(...)` results are `True`: the data read back is exactly the same as the original.

Selecting and filtering

```
# Chapter 23 - Selecting columns/rows and filtering
import pandas as pd

trees = pd.DataFrame({
    "species": ["Teak", "Meranti", "Mahogany", "Sengon", "Teak", "Meranti"],
    "plot": ["A", "A", "B", "C", "B", "C"],
    "diameter_cm": [32.5, 45.2, 28.0, 19.8, 36.1, 51.5],
    "healthy": [True, False, True, True, False, True],
})
```

```

})

print(trees[["species", "diameter_cm"]].head(2))          # several columns
print(trees.iloc[0])                                     # row number 0 by position
print(trees.loc[2, "species"])                           # row labelled 2,
column "species"

big = trees[trees["diameter_cm"] >= 30]                  # filter with a condition
print(big)

sick_big = trees[(trees["diameter_cm"] >= 30) & (~trees["healthy"])] # & = AND, ~
= NOT
print(sick_big)
print(trees.query("plot == 'A' or species == 'Teak'"))   # another way, reads like
a sentence
print(trees.sort_values("diameter_cm", ascending=False).head(3))

```

```

    species  diameter_cm
0      Teak          32.5
1  Meranti          45.2
species      Teak
plot          A
diameter_cm    32.5
healthy        True
Name: 0, dtype: object
Mahogany
    species plot  diameter_cm  healthy
0      Teak   A           32.5     True
1  Meranti   A           45.2    False
4      Teak   B           36.1    False
5  Meranti   C           51.5     True
species plot  diameter_cm  healthy
1  Meranti   A           45.2    False
4      Teak   B           36.1    False
species plot  diameter_cm  healthy
0      Teak   A           32.5     True
1  Meranti   A           45.2    False
4      Teak   B           36.1    False
species plot  diameter_cm  healthy
5  Meranti   C           51.5     True
1  Meranti   A           45.2    False
4      Teak   B           36.1    False

```

- `df[["a", "b"]]` selects columns; `iloc` selects by **position**, `loc` by **label**.
- `df[df["diameter_cm"] >= 30]` filters rows with a condition. To combine conditions use `&` (and), `|` (or), `~` (not), **with each condition in brackets**.
- `query("plot == 'A' or species == 'Teak'")` is a way that reads more like a sentence.
- `sort_values(..., ascending=False)` sorts from the largest.

Aggregation: summarising per group

```

# Chapter 23 - Aggregation: groupby, value_counts, pivot_table
import pandas as pd

trees = pd.DataFrame({
    "species": ["Teak", "Meranti", "Mahogany", "Sengon", "Teak", "Meranti"],
    "plot": ["A", "A", "B", "C", "B", "C"],
    "diameter_cm": [32.5, 45.2, 28.0, 19.8, 36.1, 51.5],

```

```

    "height_m": [18.0, 25.5, 15.0, 12.0, 20.5, 28.0],
})

print(trees["species"].value_counts())          # how many trees per species
print(trees.groupby("species")["diameter_cm"].mean().round(2))

summary = trees.groupby("plot").agg(
    count=("species", "count"),
    mean_diameter=("diameter_cm", "mean"),
    max_height=("height_m", "max"),
)
print(summary.round(2))

print(trees.pivot_table(index="plot",      columns="species",      values="diameter_cm",
aggfunc="mean"))

```

```

species
Teak      2
Meranti   2
Mahogany  1
Sengon    1
Name: count, dtype: int64

species
Mahogany   28.00
Meranti    48.35
Sengon     19.80
Teak       34.30
Name: diameter_cm, dtype: float64

   count  mean_diameter  max_height
plot
A        2           38.85         25.5
B        2           32.05         20.5
C        2           35.65         28.0

species  Mahogany  Meranti  Sengon  Teak
plot
A         NaN     45.2     NaN    32.5
B         28.0     NaN     NaN    36.1
C         NaN     51.5    19.8    NaN

```

`groupby` works like sorting sacks by species and then weighing each pile: the **split, compute, combine** pattern. The mean diameter of Teak is 34.30 (from 32.5 and 36.1), of Meranti 48.35. `agg` allows several summaries at once with result columns you name yourself. `pivot_table` makes a cross table of plot x species; cells without data are filled with `NaN` (“not a number” = empty).

Joining tables and missing data

```

# Chapter 23 - Joining tables (merge, concat) and missing data (NaN)
import pandas as pd

harvest = pd.DataFrame({"commodity": ["coffee", "rice", "catfish"], "yield_kg": [245.5,
3100.0, 432.4]})
prices = pd.DataFrame({"commodity": ["coffee", "rice", "maize"], "price_per_kg":
[58000, 6200, 5100]})

# how="left": every harvest row is kept; rows without a price partner are filled
with NaN
joined = harvest.merge(prices, on="commodity", how="left")
print(joined)
print("Rows with empty price:", joined["price_per_kg"].isna().sum())

```

```

joined["income"] = joined["yield_kg"] * joined["price_per_kg"]
print(joined)
print(joined.fillna({"price_per_kg": 0, "income": 0}))      # fill the empties with 0
print(joined.dropna())                                     # drop rows that
                                                            contain empties

extra = pd.DataFrame({"commodity": ["cattle"], "yield_kg": [1151.5]})
print(pd.concat([harvest, extra], ignore_index=True))      # stacking rows downward

```

```

commodity yield_kg price_per_kg
0  coffee    245.5    58000.0
1   rice    3100.0     6200.0
2  catfish    432.4         NaN
Rows with empty price: 1
commodity yield_kg price_per_kg income
0  coffee    245.5    58000.0 14239000.0
1   rice    3100.0     6200.0 19220000.0
2  catfish    432.4         NaN         NaN
commodity yield_kg price_per_kg income
0  coffee    245.5    58000.0 14239000.0
1   rice    3100.0     6200.0 19220000.0
2  catfish    432.4         0.0         0.0
commodity yield_kg price_per_kg income
0  coffee    245.5    58000.0 14239000.0
1   rice    3100.0     6200.0 19220000.0
commodity yield_kg
0  coffee    245.5
1   rice    3100.0
2  catfish    432.4
3   cattle   1151.5

```

merge joins two tables through a shared column, like JOIN in a database (Chapter 24). With `how="left"`, all rows of `harvest` are kept; catfish has no price so it is `NaN`, and its `income` is empty too. There are three ways to handle empties: leave them, `fillna` (fill a replacement value), or `dropna` (drop the row). **Choose according to what the data means:** filling the catfish price with 0 means “free”, which is not necessarily true.

TIP

After a `merge`, always check `isna().sum()`. Unpaired rows are almost always a sign of a name that does not match (for example `coffee` versus `Coffee`) or of data that simply does not exist yet.

Running Project I2: merging five sectors, weather, and prices

Step 3: ten harvest records from five sectors (forestry, plantation, cropland, fisheries, livestock), the Bogor weather, and the prices are merged into one table. (Fictional data; copied into the script so it stands on its own, with the same shape as the Chapter 21 results.)

```

# Chapter 23 - Running project I2, step 3: merging five sectors, weather, and prices
import pandas as pd

# Five-sector data (fictional). Copied here so the script stands on its own.
harvest = pd.DataFrame(
    [
        ("2026-01-01", "Forestry", "Plot A", "teak", 1200.0),
        ("2026-01-01", "Plantation", "Block A", "coffee", 245.5),
        ("2026-01-02", "Plantation", "Block B", "cocoa", 198.5),
        ("2026-01-02", "Cropland", "Paddy 1", "rice", 3100.0),
        ("2026-01-03", "Cropland", "Paddy 2", "maize", 1850.0),
    ]
)

```

```

        ("2026-01-03", "Fisheries", "Pond A", "catfish", 432.4),
        ("2026-01-04", "Livestock", "Barn 1", "cattle", 1151.5),
        ("2026-01-04", "Plantation", "Block C", "coffee", 310.0),
        ("2026-01-05", "Cropland", "Paddy 1", "rice", 2950.0),
        ("2026-01-05", "Plantation", "Block D", "cocoa", 276.0)],
        columns=["date", "sector", "unit", "commodity", "yield_kg"],
    )
    # Bogor weather and prices: same shape as the data from the mock server in Chapter
    21 (fictional)
    weather = pd.DataFrame({
        "date": ["2026-01-01", "2026-01-02", "2026-01-03", "2026-01-04", "2026-01-05"],
        "max_temp": [30.1, 31.4, 29.8, 28.9, 30.6],
        "rain_mm": [12.5, 0.0, 8.2, 25.4, 3.1],
    })
    prices = pd.DataFrame(
        [(d, c, p) for c, values in {
            "rice": [6200, 6250, 6180, 6300, 6270],
            "maize": [5100, 5150, 5200, 5120, 5180],
            "coffee": [58000, 58500, 59000, 57500, 58200],
            "cocoa": [62000, 61500, 62500, 63000, 62800]}.items()
         for d, p in zip(weather["date"], values)],
        columns=["date", "commodity", "price_per_kg"],
    )

    merged = (harvest
        .merge(weather, on="date", how="left")
        .merge(prices, on=["date", "commodity"], how="left"))
    merged["income_rp"] = merged["yield_kg"] * merged["price_per_kg"]

    print(merged.drop(columns=["date"]).to_string(index=False))
    print()
    print("Rows without a price (forestry, fish, cattle):",
    int(merged["price_per_kg"].isna().sum()))
    per_sector = merged.groupby("sector").agg(yield_kg=("yield_kg", "sum"),
    income_rp=("income_rp", "sum"))
    print(per_sector.round(0))

```

	sector	unit	commodity	yield_kg	max_temp	rain_mm	price_per_kg	income_rp
	Forestry	Plot A	teak	1200.0	30.1	12.5	NaN	NaN
	Plantation	Block A	coffee	245.5	30.1	12.5	58000.0	14239000.0
	Plantation	Block B	cocoa	198.5	31.4	0.0	61500.0	12207750.0
	Cropland	Paddy 1	rice	3100.0	31.4	0.0	6250.0	19375000.0
	Cropland	Paddy 2	maize	1850.0	29.8	8.2	5200.0	9620000.0
	Fisheries	Pond A	catfish	432.4	29.8	8.2	NaN	NaN
	Livestock	Barn 1	cattle	1151.5	28.9	25.4	NaN	NaN
	Plantation	Block C	coffee	310.0	28.9	25.4	57500.0	17825000.0
	Cropland	Paddy 1	rice	2950.0	30.6	3.1	6270.0	18496500.0
	Plantation	Block D	cocoa	276.0	30.6	3.1	62800.0	17332800.0

```

Rows without a price (forestry, fish, cattle): 3
      yield_kg  income_rp

```

sector	yield_kg	income_rp
Cropland	7900.0	47491500.0
Fisheries	432.0	0.0
Forestry	1200.0	0.0
Livestock	1152.0	0.0
Plantation	1030.0	61604550.0

Trace the result:

- Each harvest record receives the temperature and rain of **the same day** (`merge` on `date`) and the price of **the same commodity and day** (`merge` on two columns).
- **3 rows without a price**: teak timber, catfish, cattle, because the price API only holds rice, maize, coffee, cocoa. `income_rp` for those rows is `NaN`.
- Summary per sector: Plantation 1030.0 kg with income Rp 61,604,550; Cropland 7900.0 kg with Rp 47,491,500. Forestry, Fisheries, and Livestock show **0** income only because `sum` ignores `NaN`, **not** because their value is really zero. Data limits like this must be stated frankly when reporting.

Exercises

1. From the array `np.array([480, 500, 510, 520, 530])`, compute the mean weight and select those heavier than 505 with a mask.
2. Make a fish DataFrame with the columns `pond`, `species`, `count`, `avg_weight_g`, then compute the biomass of each row (`count * avg_weight_g / 1000`) as a new column.
3. In `06_aggregation.py`, add an aggregation of the number of trees and the smallest diameter per species.
4. **Challenge**: in `08_project_merge.py`, compute the mean income per sector **only from rows that have a price**, and compute what percentage of rows have no price.

Chapter Summary

- **numpy array**: one kind of content, operations **all at once** without `for`; `axis=0` per column, `axis=1` per row.
- **DataFrame** = a labelled table; one column = a **Series**. `head`, `describe`, `dtypes`, `shape` help you get to know it.
- Read/write: `read_csv`, `read_excel`, `to_json`, and friends; `index=False` avoids the row-number column.
- Filter with conditions (`&`, `|`, `~`, each in brackets) or `query`; sort with `sort_values`.
- `groupby` + `agg` summarises per group; `merge` joins tables; beware of `NaN` after merging.
- Libraries can have different defaults (numpy vs pandas standard deviation).

Next, in Chapter 24: that merged data vanishes when the program is closed. We store it in a **database**: SQLite, SQLAlchemy, and PostgreSQL.

Chapter 24 — Databases: A Filing Cabinet That Never Forgets

Series I2 — Data and the Internet

In this chapter, you will:

- Understand what a **database**, a table, a row, and **SQL** are
- Use **SQLite** through Python's built-in `sqlite3` module
- Write `INSERT`, `SELECT`, `UPDATE`, `DELETE`, `JOIN`, and `GROUP BY`, and avoid **SQL injection**
- Meet **SQLAlchemy** (Core and ORM) and how to connect to **PostgreSQL**
- Store the merged data of Project I2 in a database

All scripts for this chapter are in the `code/ch24/` folder. On the web version, you can also [download them all at once \(zip\)](#).

A CSV file (Chapter 23) is like a pile of note papers: easy to make, but hard to search, and it can turn into a mess when two people write at the same time. A **database** is a **filing cabinet with drawers**: each drawer (**table**) has fixed columns, each sheet (**row**) follows the rules, and the filing clerk (**SQL**, *Structured Query Language*) answers questions like “what is the total plantation harvest?” without you turning over every sheet. It also keeps storage **safe**: a change that is half done can be cancelled.

Three tables linked through shared columns

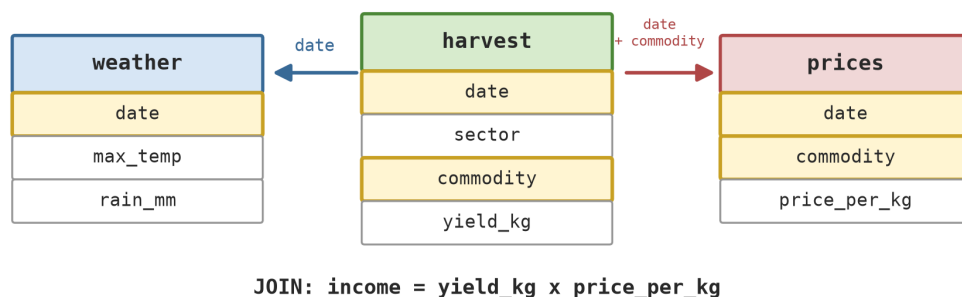


Figure 0.1: Three tables (harvest, weather, prices) linked through shared columns. This is the heart of a relational database.

SQLite: a database in one file

SQLite is a database engine that lives as **one file** and comes with Python (the `sqlite3` module), so there is nothing to install and no server. It suits learning, small applications, and data you can carry around.

```
# Chapter 24 - SQLite with the sqlite3 module (built into Python)
import sqlite3
from pathlib import Path

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
db_file = tmp / "field.db"
```

```

db_file.unlink(missing_ok=True)                # start from an empty database so the
result is the same

connection = sqlite3.connect(db_file)          # the file is created if it does not exist
connection.execute("""
    CREATE TABLE tree (
        id INTEGER PRIMARY KEY,
        species TEXT NOT NULL,
        plot TEXT NOT NULL,
        diameter_cm REAL
    )
""")
data = [("Teak", "A", 32.5), ("Meranti", "A", 45.2), ("Mahogany", "B", 28.0), ("Teak",
"B", 36.1)]
connection.executemany("INSERT INTO tree (species, plot, diameter_cm) VALUES
(?, ?, ?)", data)
connection.commit()                            # without commit, changes are not
saved permanently

for row in connection.execute("SELECT id, species, plot, diameter_cm FROM tree ORDER
BY diameter_cm DESC"):
    print(row)

count = connection.execute("SELECT COUNT(*) FROM tree").fetchone()[0]
print("Number of trees:", count)
connection.close()

```

```

(2, 'Meranti', 'A', 45.2)
(4, 'Teak', 'B', 36.1)
(1, 'Teak', 'A', 32.5)
(3, 'Mahogany', 'B', 28.0)
Number of trees: 4

```

- `CREATE TABLE` defines the columns and their kinds; `PRIMARY KEY` = a unique number for each row; `NOT NULL` = must be filled.
- `?` is a **value placeholder**; the values are given separately. `executemany` inserts many rows at once.
- `commit()` saves the changes permanently. Without a commit, changes may be lost.
- `ORDER BY diameter_cm DESC` sorts from large to small: Meranti 45.2 is on top.

Never paste user text into SQL

```

# Chapter 24 - Parameters (?) against SQL injection
import sqlite3

connection = sqlite3.connect(":memory:")        # a temporary database in memory
connection.execute("CREATE TABLE officer (name TEXT, area TEXT)")
connection.executemany("INSERT INTO officer VALUES (?, ?)", [("Sari", "Plot A"),
("Budi", "Plot B")])

user_input = "x' OR '1'='1"                    # a mischievous entry from a user

# WRONG WAY: pasting user text straight into the SQL (f-string)
bad_sql = f"SELECT name FROM officer WHERE name = '{user_input}'"
print("SQL that was built:", bad_sql)
print("Wrong way result :", connection.execute(bad_sql).fetchall())    # leak: every
row comes out!

# RIGHT WAY: question mark + separate value; the text is treated as DATA, not as

```

```
a command
print("Right way result :", connection.execute("SELECT name FROM officer WHERE name
= ?", (user_input,)).fetchall())
```

```
SQL that was built: SELECT name FROM officer WHERE name = 'x' OR '1'='1'
Wrong way result : [('Sari',), ('Budi',)]
Right way result : []
```

⚠ WARNING

In the wrong way, the mischievous entry `x' OR '1'='1'` becomes **part of the SQL command** and makes the condition always true, so **every row leaks**. This is called **SQL injection**, one of the most common attacks in the world. The right way: always use `?` (or `:name`) and give the values separately; the text is treated as **data**, never as a command. The result: no row matches.

Update, delete, and transactions

```
# Chapter 24 - UPDATE, DELETE, and transactions (commit / rollback)
import sqlite3

connection = sqlite3.connect(":memory:")
connection.execute("CREATE TABLE pond (name TEXT PRIMARY KEY, fish INTEGER CHECK (fish
>= 0))")
connection.executemany("INSERT INTO pond VALUES (?, ?)", [("A", 800), ("B", 650),
("C", 400)])
connection.commit()

connection.execute("UPDATE pond SET fish = fish - 50 WHERE name = 'A'")          # 50
fish harvested
connection.execute("DELETE FROM pond WHERE name = 'C'")                        # pond
C dismantled
print(connection.execute("SELECT * FROM pond ORDER BY name").fetchall())
connection.commit()                    # save first: a safe point before
the next attempt

# Transaction: all steps succeed together, or are cancelled together
try:
    with connection:                    # automatic commit; rollback on
an error
        connection.execute("UPDATE pond SET fish = fish - 100 WHERE name = 'B'")
        connection.execute("UPDATE pond SET fish = fish - 900 WHERE name = 'A'")  #
breaks the CHECK
except sqlite3.IntegrityError as error:
    print("Cancelled:", error)

print("After rollback:", connection.execute("SELECT * FROM pond ORDER BY
name").fetchall())
```

```
[('A', 750), ('B', 650)]
Cancelled: CHECK constraint failed: fish >= 0
After rollback: [('A', 750), ('B', 650)]
```

After harvesting 50 fish, pond A has 750 left; pond C is dismantled. Then comes a **transaction**: two changes that must succeed **together**. The second change (take 900 fish from A) breaks the rule `CHECK (fish >= 0)`, so SQLite refuses with an `IntegrityError`, and the `with connection:` block **Cancels** both changes (*rollback*): pond B stays at 650, not 550. That is the use of a transaction: no half-finished state.

i REMEMBER

A database can guard data rules for you through **constraints** (NOT NULL, UNIQUE, CHECK, PRIMARY KEY, FOREIGN KEY). Let the database refuse bad data; do not rely on Python code alone.

Two tables: JOIN and GROUP BY

```
# Chapter 24 - Two tables: JOIN and GROUP BY
import sqlite3

connection = sqlite3.connect(":memory:")
connection.executescript("""
    CREATE TABLE sector (id INTEGER PRIMARY KEY, name TEXT);
    CREATE TABLE harvest (id INTEGER PRIMARY KEY, sector_id INTEGER REFERENCES
sector(id),
                        commodity TEXT, yield_kg REAL);
    INSERT INTO sector VALUES (1, 'Plantation'), (2, 'Cropland'), (3, 'Fisheries');
    INSERT INTO harvest (sector_id, commodity, yield_kg) VALUES
        (1, 'coffee', 245.5), (1, 'cocoa', 198.5), (1, 'coffee', 310.0),
        (2, 'rice', 3100.0), (2, 'maize', 1850.0), (3, 'catfish', 432.4);
""")

sql = """
        SELECT  sector.name AS sector,      COUNT(*) AS harvest_count,
ROUND(SUM(harvest.yield_kg), 1) AS total_kg
        FROM harvest
        JOIN sector ON sector.id = harvest.sector_id
        GROUP BY sector.name
        HAVING SUM(harvest.yield_kg) > 500
        ORDER BY total_kg DESC
    """

for row in connection.execute(sql):
    print(row)
```

```
('Cropland', 2, 4950.0)
('Plantation', 3, 754.0)
```

JOIN connects the rows of two tables where `sector_id` equals the sector's `id`, so the sector name is shown. GROUP BY groups and SUM/COUNT summarise, exactly like pandas `groupby` in Chapter 23. HAVING filters **after** grouping (only sectors with a total above 500 kg), whereas WHERE filters **before**. Cropland: 2 harvests, 4950.0 kg (3100 + 1850); Plantation: 3 harvests, 754.0 kg; Fisheries is dropped by HAVING because it has only 432.4 kg.

pandas and SQL

```
# Chapter 24 - pandas and SQL exchange data
import sqlite3
import pandas as pd

connection = sqlite3.connect(":memory:")

ponds = pd.DataFrame({"pond": ["A", "B", "C"], "species": ["catfish", "tilapia",
"catfish"],
                    "fish": [800, 650, 400], "weight_g": [540.5, 320.0, 480.0]})
ponds.to_sql("pond", connection, index=False) # DataFrame -> SQL table
```

```

result = pd.read_sql_query(
    "SELECT species, SUM(fish) AS total_fish, ROUND(AVG(weight_g), 1) AS avg_weight_g
    FROM pond GROUP BY species",
    connection,
)
# SQL result -> DataFrame
print(result)

heavy = pd.read_sql_query("SELECT * FROM pond WHERE weight_g > ?", connection,
params=(450,))
print(heavy)

```

```

species total_fish avg_weight_g
0 catfish      1200        510.3
1 tilapia       650        320.0
pond species fish weight_g
0 A catfish   800    540.5
1 C catfish   400    480.0

```

`to_sql` turns a DataFrame into a table; `read_sql_query` turns the result of a `SELECT` into a DataFrame. The two tools are friends: SQL is good for sifting large data inside the database, pandas is good for analysing the result.

SQLAlchemy: one way into many databases

Writing SQL directly ties you to one brand of database. **SQLAlchemy** is a library (`pip install sqlalchemy`) that acts as a translator: the same code can run on SQLite or PostgreSQL, you only change the **connection address** (URL) in `create_engine`. It has two layers.

Core: you still write SQL, but with safe named parameters `:name` and tidy connection handling (`engine.begin()` = automatic commit at the end of the block):

```

# Chapter 24 - SQLAlchemy: engine and text() (the "Core" layer)
from pathlib import Path
from sqlalchemy import create_engine, text

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
db_file = tmp / "sqlalchemy_core.db"
db_file.unlink(missing_ok=True)

engine = create_engine(f"sqlite:/// {db_file.as_posix()}") # URL: database_kind:/// path

with engine.begin() as connection: # begin(): automatic
    commit at the end of the block
    connection.execute(text("CREATE TABLE livestock (barn TEXT, species TEXT, weight_kg
    REAL)"))
    connection.execute(
        text("INSERT INTO livestock VALUES (:barn, :species, :weight)"), # :name
        = named parameter
        [{"barn": "K1", "species": "cattle", "weight": 350.0},
        {"barn": "K1", "species": "cattle", "weight": 412.5},
        {"barn": "K2", "species": "cattle", "weight": 389.0}],
    )

with engine.connect() as connection:
    result = connection.execute(text("SELECT barn, AVG(weight_kg) AS mean FROM livestock
    GROUP BY barn"))
    for row in result:
        print(row.barn, round(row.mean, 2))

```

```
print("dialect:", engine.dialect.name)
engine.dispose() # release the file
```

```
K1 381.25
K2 389.0
dialect: sqlite
```

ORM (*Object-Relational Mapper*): tables are written as **Python classes** (remember OOP, Chapter 16), rows become **objects**, and relationships between tables become attributes:

```
# Chapter 24 - SQLAlchemy ORM: tables as Python classes
from pathlib import Path
from sqlalchemy import ForeignKey, create_engine, func, select
from sqlalchemy.orm import DeclarativeBase, Mapped, Session, mapped_column,
relationship

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
db_file = tmp / "sqlalchemy_orm.db"
db_file.unlink(missing_ok=True)

class Base(DeclarativeBase):
    pass

class Sector(Base):
    __tablename__ = "sector"
    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str]
    harvests: Mapped[list["Harvest"]] = relationship(back_populates="sector")

class Harvest(Base):
    __tablename__ = "harvest"
    id: Mapped[int] = mapped_column(primary_key=True)
    commodity: Mapped[str]
    yield_kg: Mapped[float]
    sector_id: Mapped[int] = mapped_column(ForeignKey("sector.id"))
    sector: Mapped[Sector] = relationship(back_populates="harvests")

engine = create_engine(f"sqlite:/// {db_file.as_posix()}")
Base.metadata.create_all(engine) # creates the tables from the
class definitions

with Session(engine) as session:
    plantation = Sector(name="Plantation", harvests=[Harvest(commodity="coffee",
yield_kg=245.5),
                                                    Harvest(commodity="cocoa", yield_kg=198.5)])
    cropland = Sector(name="Cropland", harvests=[Harvest(commodity="rice",
yield_kg=3100.0)])
    session.add_all([plantation, cropland])
    session.commit()

with Session(engine) as session:
    for sector in session.scalars(select(Sector).order_by(Sector.name)):
        print(sector.name, "->", [h.commodity for h in sector.harvests]) #
relationship, automatically
```

```

                                                    biggest
= session.scalars(select(Harvest).order_by(Harvest.yield_kg.desc())).first()
    print("Biggest harvest:", biggest.commodity, biggest.yield_kg, "kg from",
biggest.sector.name)
    print("Total kg:", session.scalar(select(func.sum(Harvest.yield_kg))))
engine.dispose()

```

```

Cropland -> ['rice']
Plantation -> ['coffee', 'cocoa']
Biggest harvest: rice 3100.0 kg from Cropland
Total kg: 3544.0

```

`Sector.harvests` holds a list of `Harvest` objects without you writing a `JOIN`; in the other direction `biggest.sector.name` travels upward. The total 3544.0 kg = 245.5 + 198.5 + 3100.0.

PostgreSQL: a server-level database

PostgreSQL is a database that runs as a **server** used by many people at once, with far richer features (users and permissions, large capacity, extensions such as PostGIS for maps in Series A2). With SQLAlchemy, the biggest difference is simply the **connection address**, shaped like `postgresql+psycopg://USER:PASSWORD@SERVER:5432/DATABASE_NAME`, which uses the `psycopg` version 3 library (`pip install "psycopg[binary]"`).

```

# Chapter 24 - Connecting to PostgreSQL (a temporary server started by the book-
building tool)
import os
from sqlalchemy import create_engine, text

# The connection address is read from an environment variable, so the password is not
written in code.
# The shape of the address on your own server:
# postgresql://USER:PASSWORD@SERVER:5432/DATABASE_NAME
# Example value of the variable (password masked):
# DATABASE_URL = postgresql://officer:[disamarkan]@localhost:5432/forest_farm
address = os.environ["DATABASE_URL"]

# SQLAlchemy picks the driver from the start of the address; "+psycopg" asks for psycopg
version 3.
address = address.replace("postgresql://", "postgresql+psycopg://", 1)
engine = create_engine(address)

with engine.begin() as connection:
    connection.execute(text("""
        CREATE TABLE IF NOT EXISTS harvest (
            id SERIAL PRIMARY KEY,
            commodity TEXT NOT NULL,
            yield_kg NUMERIC(10, 1)
        )
    """))
    connection.execute(text("INSERT INTO harvest (commodity, yield_kg) VALUES
(:c, :y)", {"c": "coffee", "y": 245.5}))

with engine.connect() as connection:
    print(connection.execute(text("SELECT commodity, yield_kg FROM
harvest")).fetchall())

```

```
[('coffee', Decimal('245.5'))]
```

This code was **really run** when the book was built, on a **temporary** PostgreSQL 14.5 started by the book-building tool: that server lives only while the book is built, can only be reached from the author's computer,

has no password, and is then deleted. A reader's or an office's own server usually has a user name and password; that part was **not tested** here. Notice two things in the output. First, the `NUMERIC` column comes back as `Decimal('245.5')`, not a `float` as in SQLite. Second, the line `address.replace(...)` adds `+psycopg` to the start of the address so SQLAlchemy uses the psycopg 3 driver.

To try it on your own computer, set the environment variable first (a PowerShell example, password masked): `$env:DATABASE_URL = "postgresql://officer:[disamarkan]@localhost:5432/forest_farm"`. Notice two safe habits: the address is read from the environment variable `DATABASE_URL` (no password in code), and the example value is written `[disamarkan]` (masked).

TECHNICAL STUFF (feel free to skip)

Differences between SQLite and PostgreSQL you will meet: PostgreSQL is strict about column types (a `NUMERIC` column comes back as `Decimal`, not `float`; this is **proven** by the output above), uses `SERIAL` or `GENERATED ... AS IDENTITY` for automatic numbers (`SERIAL` is used and works in the example), and a table created inside a transaction becomes visible to others after the commit. SQLite is looser about column types. That is why you test on the engine you will really use. (The points about `GENERATED ... AS IDENTITY` and transaction visibility are general knowledge that has **not been tested** in this book.)

Running Project I2: storing the merged data

Step 4: the fictional data is stored as **three separate tables** (harvest, weather, prices) in an SQLite file, like a real database, then one SQL query with two `LEFT JOINs` computes the total yield, income, and mean rain per sector.

```
# Chapter 24 - Running project I2, step 4: saving the merged data in a database
from pathlib import Path
import pandas as pd
from sqlalchemy import create_engine, text

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
db_file = tmp / "project_i2.db"
db_file.unlink(missing_ok=True)

# The (fictional) data is copied so the script stands on its own; same shape as Chapters
21 and 23.
harvest = pd.DataFrame(
    [
        ("2026-01-01", "Forestry", "Plot A", "teak", 1200.0),
        ("2026-01-01", "Plantation", "Block A", "coffee", 245.5),
        ("2026-01-02", "Plantation", "Block B", "cocoa", 198.5),
        ("2026-01-02", "Cropland", "Paddy 1", "rice", 3100.0),
        ("2026-01-03", "Cropland", "Paddy 2", "maize", 1850.0),
        ("2026-01-03", "Fisheries", "Pond A", "catfish", 432.4),
        ("2026-01-04", "Livestock", "Barn 1", "cattle", 1151.5),
        ("2026-01-04", "Plantation", "Block C", "coffee", 310.0),
        ("2026-01-05", "Cropland", "Paddy 1", "rice", 2950.0),
        ("2026-01-05", "Plantation", "Block D", "cocoa", 276.0),
    ],
    columns=["date", "sector", "unit", "commodity", "yield_kg"])
dates = ["2026-01-01", "2026-01-02", "2026-01-03", "2026-01-04", "2026-01-05"]
weather = pd.DataFrame(
    {"date": dates, "max_temp": [30.1, 31.4, 29.8, 28.9, 30.6],
     "rain_mm": [12.5, 0.0, 8.2, 25.4, 3.1]})
price_lists = {"rice": [6200, 6250, 6180, 6300, 6270], "maize": [5100, 5150, 5200, 5120, 5180],
               "coffee": [58000, 58500, 59000, 57500, 58200], "cocoa": [62000, 61500, 62500, 63000, 62800]}
prices = pd.DataFrame([(d, c, p) for c, v in price_lists.items() for d, p in zip(dates, v)],
                      columns=["date", "commodity", "price_per_kg"])
```

```

engine = create_engine(f"sqlite:///db_file.as_posix()")
harvest.to_sql("harvest", engine, index=False)      # three separate tables, like a
real database
weather.to_sql("weather", engine, index=False)
prices.to_sql("prices", engine, index=False)

sql = text("""
    SELECT h.sector,
           ROUND(SUM(h.yield_kg), 1) AS yield_kg,
           ROUND(SUM(h.yield_kg * p.price_per_kg), 0) AS income_rp,
           ROUND(AVG(w.rain_mm), 1) AS mean_rain_mm
    FROM harvest h
    LEFT JOIN prices p ON p.date = h.date AND p.commodity = h.commodity
    LEFT JOIN weather w ON w.date = h.date
    GROUP BY h.sector
    ORDER BY h.sector
""")
with engine.connect() as connection:
    for table in ["harvest", "weather", "prices"]:
        n = connection.execute(text(f"SELECT COUNT(*) FROM {table}")).scalar()
        print(f"table {table:7}: {n} rows")
        print(pd.read_sql_query(sql, connection).to_string(index=False))
engine.dispose()

```

```

table harvest: 10 rows
table weather: 5 rows
table prices : 20 rows

```

sector	yield_kg	income_rp	mean_rain_mm
Cropland	7900.0	47491500.0	3.8
Fisheries	432.4	NaN	8.2
Forestry	1200.0	NaN	12.5
Livestock	1151.5	NaN	25.4
Plantation	1030.0	61604550.0	10.3

The result agrees with pandas in Chapter 23: Plantation 1030.0 kg and Rp 61,604,550; Cropland 7900.0 kg and Rp 47,491,500. Sectors without a price show `NaN` for income because a `SUM` over empty values stays empty. Unlike pandas, which sums empties to 0. **Mean rain per sector** is the mean of the rain on that sector's harvest days (not rain for a month), so interpret it with care.

Exercises

1. In `01_sqlite_basics.py`, add one new tree, then print only the trees with `plot = 'B'` (use `WHERE plot = ?`).
2. In `03_update_delete_transaction.py`, change the deduction from A to 100 so it does not break the rule. What happens to the whole transaction?
3. In `04_join_aggregate.py`, remove the `HAVING` and add `ORDER BY sector.name`. Which sectors appear?
4. **Challenge:** in `07_sqlalchemy_orm.py`, add the sector "Fisheries" with one harvest "catfish" 432.4 kg, then print the total yield per sector with `func.sum` and `group_by`.

Chapter Summary

- A **database** = a filing cabinet with drawers: tables, rows, columns; **SQL** = the language to ask and change.
- **SQLite** = one file, built into Python (`sqlite3`); save changes with `commit()`.
- Always use parameters (`?, :name`), **never** paste user text into SQL (SQL injection).
- A **transaction** makes a group of changes succeed together or cancel together; constraints (`CHECK`, `NOT NULL`, ...) guard data quality.
- **JOIN** connects tables; **GROUP BY** + **SUM/COUNT** summarise; **WHERE** filters before, **HAVING** after grouping.

- **SQLAlchemy** (Core and ORM) unifies access to many databases; **PostgreSQL** only needs a different connection address, and that part is run in this book on a temporary server (your own server still needs a user name and password).

Next, in Chapter 25: the data is stored neatly. Time to analyse it: advanced pandas, numpy, matplotlib, scikit-learn, and Jupyter Notebook, then closing Series I2.

Chapter 25 — Data Science and Analysis: From Tables to Pictures and Guesses

Series I2 — Data and the Internet

In this chapter, you will:

- Use advanced pandas: derived columns, categories, and time-ordered data
- Use numpy for statistics, `np.where`, and **broadcasting**
- Draw charts with **matplotlib**
- Build your first models with **scikit-learn**: regression and classification, tested on data the model has not seen
- Meet **Jupyter Notebook** and **Anaconda**, then close Running Project I2

All scripts for this chapter are in the `code/ch25/` folder. On the web version, you can also [download them all at once \(zip\)](#).

The data came in (Chapter 21), was cleaned (22), arranged (23), and stored (24). Now comes the typical data scientist's question: "What can we see and guess from this data?" Imagine you are a plantation foreman who has kept two months of rain and coffee harvest notes. A good foreman does four things: **inspects** (summary numbers), **draws** (charts), **guesses** (a model), and **tests** the guess (is it still right for new data?). This chapter tries all four. All data here is **fictional**, made with random numbers on a fixed seed (`default_rng(42)`) so the result is always the same; because of that, the formula that made the data is known and we can check whether the model finds it again.

Advanced pandas: derived columns and categories

```
# Chapter 25 - Advanced pandas: new columns, categories, and summaries
import numpy as np
import pandas as pd

rng = np.random.default_rng(42) # fixed seed: the result is
always the same
n = 60
data = pd.DataFrame({
    "date": pd.date_range("2026-01-01", periods=n),
    "rain_mm": rng.gamma(shape=1.2, scale=8.0, size=n).round(1),
})
data["coffee_yield_kg"] = (200 + 1.5 * data["rain_mm"].clip(upper=30) + rng.normal(0,
8, n)).round(1)

# pd.cut: turns numbers into categories using boundaries
data["category"] = pd.cut(data["rain_mm"], bins=[-0.1, 5, 15, 1000], labels=["dry",
"moderate", "heavy"])
print(data.head(5))
print(data["category"].value_counts())

summary = data.groupby("category", observed=True)["coffee_yield_kg"].agg(["count",
"mean", "max"]).round(1)
print(summary)

data["above_mean"] = data["coffee_yield_kg"] > data["coffee_yield_kg"].mean() # a
True/False column
```

```
print("Days above the mean:", int(data["above_mean"].sum()), "out of", n)
print(data.nlargest(3, "coffee_yield_kg")[["date", "rain_mm", "coffee_yield_kg"]])
```

```

      date  rain_mm  coffee_yield_kg  category
0 2026-01-01      9.5           220.9  moderate
1 2026-01-02     14.2           224.2  moderate
2 2026-01-03      7.9           223.6  moderate
3 2026-01-04      6.8           200.7  moderate
4 2026-01-05     15.8           218.6    heavy

category
moderate    29
dry         24
heavy        7
Name: count, dtype: int64

      count    mean    max
category
dry         24  201.5  221.7
moderate    29  216.5  242.2
heavy        7  227.7  242.3
Days above the mean: 30 out of 60
      date  rain_mm  coffee_yield_kg
57 2026-02-27     27.0           242.3
15 2026-01-16     12.6           242.2
38 2026-02-08     12.7           236.1
```

- A new column is made by assigning the result of a formula: `data["coffee_yield_kg"] = ...`
- `pd.cut` turns numbers into categories using boundaries: rain 0 to 5 mm = dry, up to 15 = moderate, above = heavy. Of 60 days: 24 dry, 29 moderate, 7 heavy.
- `groupby(...).agg(...)` gives the mean coffee yield per category: **201.5 kg** (dry), **216.5 kg** (moderate), and **227.7 kg** (heavy). The rising pattern is clear.
- `nlargest(3, ...)` takes the three biggest harvest days; `.sum()` on a True/False column counts the True values (30 days above the mean).

Time-ordered data

```

# Chapter 25 - Time-ordered data: date index, resample, rolling
import numpy as np
import pandas as pd

rng = np.random.default_rng(42)
rain = pd.Series(rng.gamma(1.2, 8.0, 28).round(1), index=pd.date_range("2026-01-01",
periods=28), name="rain_mm")

print(rain.loc["2026-01-03":"2026-01-05"])           # selecting by a range of dates
weekly = rain.resample("W").sum().round(1)           # total per week (a week ends
on Sunday)
print(weekly)

moving = rain.rolling(window=7).mean().round(2)      # mean of the last 7 days
print(moving.iloc[[5, 6, 7, 27]])                  # the first 6 days are still
empty (NaN)
print("Change from yesterday, first 3 days:", rain.diff().iloc[1:4].round(1).tolist())
```

```

2026-01-03      7.9
2026-01-04      6.8
2026-01-05     15.8
Freq: D, Name: rain_mm, dtype: float64
2026-01-04     38.4
```

```

2026-01-11    86.2
2026-01-18    51.3
2026-01-25    64.7
2026-02-01     9.9
Freq: W-SUN, Name: rain_mm, dtype: float64
2026-01-06     NaN
2026-01-07    10.37
2026-01-08    10.46
2026-01-28     8.00
Name: rain_mm, dtype: float64
Change from yesterday, first 3 days: [4.7, -6.3, -1.1]

```

When the index is made of dates, you can **slice by a range of dates** (`loc["2026-01-03":"2026-01-05"]`), **summarise per period** (`resample("W")` = per week), and compute a **moving average** (`rolling(7).mean()`). The first six days are empty (NaN) because there are not yet seven days to average; day 7 (7 January) is 10.37. Notice that the first week (4 days) and the last week (ending 1 February, only 3 days of data) are not full; the last week's total is small (9.9 mm) because of that. Incomplete periods can mislead.

Advanced numpy

```

# Chapter 25 - Advanced numpy: statistics, np.where, and broadcasting
import numpy as np

rng = np.random.default_rng(42)
rain = rng.gamma(1.2, 8.0, 30).round(1)
temp = (28 + 0.05 * rain + rng.normal(0, 1, 30)).round(1)

print("percentiles 25/50/75:", np.percentile(rain, [25, 50, 75]).round(2))
print("wettest day: day", int(rain.argmax()) + 1, "with", rain.max(), "mm")
print("rain-temperature correlation:", np.corrcoef(rain, temp)[0, 1].round(3))

label = np.where(rain > 10, "wet", "dry")          # if-else for a whole array
print("number of wet days:", int((label == "wet").sum()), "out of", label.size)

cumulative = np.cumsum(rain)                      # running total
print("cumulative rain on day 30:", cumulative[-1].round(1))

# Broadcasting: a 2-D array minus a 1-D array (per column), with no loop
table = np.column_stack([rain, temp])
z = (table - table.mean(axis=0)) / table.std(axis=0)    # standardising: mean 0,
spread 1
print("mean after standardising:", z.mean(axis=0).round(6), "| spread:",
z.std(axis=0).round(6))

percentiles 25/50/75: [ 4.05  7.85 12.38]
wettest day: day 11 with 20.6 mm
rain-temperature correlation: 0.233
number of wet days: 12 out of 30
cumulative rain on day 30: 253.2
mean after standardising: [-0. -0.] | spread: [1. 1.]

```

- `np.percentile` gives boundary values: half of the days have rain below 7.85 mm (the median).
- `argmax()` gives the **position** of the largest value (day 11, 20.6 mm); because positions start at 0, we add 1 for the day number.
- `np.corrcoef` computes the **correlation** (0.233: weak). Correlation only measures how much two numbers move together; it is **not** proof of cause and effect. Series I3 covers it in depth.
- `np.where(condition, a, b)` is an if-else for a whole array. 12 of 30 days are “wet”.

- **Broadcasting:** `table - table.mean(axis=0)` subtracts each column's mean from its own column with no loop. After standardising, the mean becomes 0 and the spread 1 (the `-0.` is just a tiny negative zero from rounding).

matplotlib: drawing

matplotlib (`pip install matplotlib`) is the most basic charting library in Python. Two ideas: the **figure** (the canvas) and the **axes** (one drawing area on that canvas).

```
# Chapter 25 - matplotlib: line and bar charts
from pathlib import Path
import matplotlib
matplotlib.use("Agg")                                # draw to a file, without
a window
import matplotlib.pyplot as plt
import numpy as np

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

rng = np.random.default_rng(42)
day = np.arange(1, 31)
rain = rng.gamma(1.2, 8.0, 30).round(1)

fig, (top, bottom) = plt.subplots(2, 1, figsize=(8, 6), sharex=True)
top.plot(day, rain, marker="o", color="tab:blue")
top.set_ylabel("Rain (mm)")
top.set_title("Daily rain, January 2026 (fictional data)")
bottom.bar(day, np.cumsum(rain), color="tab:green")
bottom.set_ylabel("Cumulative rain (mm)")
bottom.set_xlabel("Day")
fig.tight_layout()
fig.savefig(tmp / "rain.png", dpi=100)
plt.close(fig)

print("Image saved:", (tmp / "rain.png").exists())
print("Number of axes:", len(fig.axes))
```

```
Image saved: True
Number of axes: 2
```

`matplotlib.use("Agg")` makes matplotlib draw to a file without opening a window (good for automated scripts). The chart is saved in `_tmp/rain.png`; the script only prints that the file exists and that the figure has 2 axes. The usual forms: `plot` (line), `bar` (bars), `scatter` (dots), then add `title`, `xlabel`, `ylabel`, and `savefig`.

💡 TIP

Every chart should answer one question, and its axes and title should state the **units**. A chart without units makes readers guess. Also add a “fictional data” note when the data is fictional.

```
# Chapter 25 - Scatter plot and trend line
from pathlib import Path
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import numpy as np

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
```

```

rng = np.random.default_rng(42)
rain = rng.uniform(0, 30, 40)
yield_kg = 200 + 1.5 * rain + rng.normal(0, 8, 40)    # true relationship: +1.5 kg per
mm (we know because we made it)

slope, intercept = np.polyfit(rain, yield_kg, deg=1) # best straight line
print(f"line: yield = {intercept:.1f} + {slope:.2f} x rain")

plt.figure(figsize=(6, 4))
plt.scatter(rain, yield_kg, alpha=0.7)
line_x = np.array([0, 30])
plt.plot(line_x, intercept + slope * line_x, color="tab:red")
plt.xlabel("Rain (mm)")
plt.ylabel("Coffee yield (kg)")
plt.title("Rain and coffee yield (fictional data)")
plt.tight_layout()
plt.savefig(tmp / "scatter.png", dpi=100)
plt.close()
print("Image saved:", (tmp / "scatter.png").exists())

```

```

line: yield = 202.6 + 1.34 x rain
Image saved: True

```

A scatter plot shows the relationship between two numbers. `np.polyfit(..., deg=1)` finds the best straight line: **yield = 202.6 + 1.34 x rain**. The data-making rule was actually $200 + 1.5 \times \text{rain}$. The difference is natural because only 40 points were given noise; the more data, the closer it gets. This is an important lesson: **a result from a small sample is only an estimate**.

scikit-learn: your first model

scikit-learn (`pip install scikit-learn`; written `sklearn` in code) is a classic machine learning library. The process is always similar: **split the data** into training and test data, **train** the model (**fit**), then **test** it on data it has never seen (**predict**).

Regression (predicting a number):

```

# Chapter 25 - scikit-learn: a first regression model (predicting a number)
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split

rng = np.random.default_rng(42)
rain = rng.uniform(0, 30, 80)
yield_kg = 200 + 1.5 * rain + rng.normal(0, 8, 80)

X = rain.reshape(-1, 1)                                # scikit-learn wants features
as a table (rows x columns)
y = yield_kg
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=42)
print("training data:", len(X_train), "| test data:", len(X_test))

model = LinearRegression()
model.fit(X_train, y_train)                             # "learns" from the training data
print(f"model: yield = {model.intercept_:.1f} + {model.coef_[0]:.2f} x rain")

predicted = model.predict(X_test)                       # the exam: data it has
never seen
print("R2 on test data:", round(r2_score(y_test, predicted), 3))

```



```
print("Mean absolute difference (kg):", round(mean_absolute_error(y_test, predicted),
2))
print("Prediction for 10 mm and 25 mm of rain:", model.predict([[10], [25]]).round(1))
```

```
training data: 60 | test data: 20
model: yield = 196.6 + 1.63 x rain
R2 on test data: 0.709
Mean absolute difference (kg): 7.42
Prediction for 10 mm and 25 mm of rain: [212.9 237.4]
```

Of 80 records, 60 were used for training and 20 for the exam. The model found **yield = 196.6 + 1.63 x rain**. An **R2** of 0.709 on the test data means the model explains roughly 71% of the variation in yield; on average it is off by about 7.42 kg. Predictions for 10 mm and 25 mm of rain: 212.9 and 237.4 kg.

i REMEMBER

Always test a model on **data that was not used for training**. A model tested only on its own training data is like a pupil examined with questions whose answers he has already memorised.

Classification (predicting a category), here healthy or sick trees:

```
# Chapter 25 - scikit-learn: classification (predicting a category) healthy or sick
trees
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

rng = np.random.default_rng(42)
n = 120
diameter = rng.uniform(15, 55, n)
height = diameter * 0.55 + rng.normal(0, 2, n)
# a fictional rule used to make the data: slender trees (tall, thin) are sick more often
sick_chance = 1 / (1 + np.exp(-(height / diameter - 0.62) * 25))
sick = rng.uniform(size=n) < sick_chance

X = np.column_stack([diameter, height])
X_train, X_test, y_train, y_test = train_test_split(X, sick, test_size=0.3,
random_state=42, stratify=sick)

tree = DecisionTreeClassifier(max_depth=3, random_state=42).fit(X_train, y_train)
print("Training accuracy:", round(accuracy_score(y_train, tree.predict(X_train)), 3))
print("Test accuracy      :", round(accuracy_score(y_test, tree.predict(X_test)), 3))
print("Baseline (always guess 'not sick'):", round(float((~y_test).mean()), 3))
print("Feature importance (diameter, height):", tree.feature_importances_.round(2))
print("Prediction for trees 20 cm/16 m and 40 cm/22 m:", tree.predict([[20, 16],
[40, 22]]))
```

```
Training accuracy: 0.893
Test accuracy      : 0.694
Baseline (always guess 'not sick'): 0.778
Feature importance (diameter, height): [0.42 0.58]
Prediction for trees 20 cm/16 m and 40 cm/22 m: [False  True]
```

The result teaches something honest. The training accuracy is 0.893, but the test accuracy is only 0.694, **lower** than the silly guess “always say not sick” (0.778). The model **memorised** the training data (*overfitting*) and is not yet useful on new data. With only 120 trees and a model as free as a decision tree, this is normal. How to improve it (more data, a simpler model, cross-validation, measures other than accuracy) belongs to Series I3 and Series A3. What matters now: **always compare a model with a simple baseline**.

Jupyter Notebook and Anaconda

Jupyter Notebook is an interactive “notebook”: text cells (Markdown) and code cells take turns, and results (numbers, tables, charts) appear right under the code cell. It is very popular for data exploration because you can try, look, and explain in one place. Its file ends in `.ipynb` and is really ordinary JSON.

Anaconda is an “all in one” bundle that brings Python, the `conda` package manager, Jupyter, and many data science libraries (installing it and comparing it with `venv` are covered in Chapter 17). On the author’s computer, the `conda` command was **not found**, so Anaconda was **not run** in this book. The notebook in this chapter was run with the Jupyter installed in the project environment (`pip`), which works the same way. You can use either: Anaconda Navigator if you want a point-and-click interface, or `pip install jupyterlab` and then `jupyter lab`.

The next script builds a notebook through code (`nbformat`), runs all its cells (`nbconvert`), then prints their output, like pressing “Run All”.

```
# Chapter 25 - Creating and running a Jupyter Notebook through code
import os
from pathlib import Path
import nbformat
from nbconvert.preprocessors import ExecutePreprocessor

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

notebook = nbformat.v4.new_notebook()
notebook.cells = [
    nbformat.v4.new_markdown_cell("# Rain notes\nA small notebook: text and code taking turns."),
    nbformat.v4.new_code_cell("import pandas as pd\nrain = pd.Series([12.5, 0.0, 8.2, 25.4, 3.1])\nrain.sum()"),
    nbformat.v4.new_code_cell("print('Mean:', rain.mean())\nrain.describe().round(2)"),
]
nbformat.write(notebook, tmp / "rain_notes.ipynb")

# Running all cells, like pressing "Run All" in Jupyter
# (The kernel's technical warnings on our screen are hidden for a moment: they are about the local connection, not errors.)
saved_stderr = os.dup(2)
os.dup2(os.open(os.devnull, os.O_WRONLY), 2)
try:
    ExecutePreprocessor(timeout=120, kernel_name="python3").preprocess(notebook, {"metadata": {"path": str(tmp)}})
finally:
    os.dup2(saved_stderr, 2)

for number, cell in enumerate(notebook.cells, start=1):
    if cell.cell_type != "code":
        continue
    print(f"--- cell {number} ---")
    for out in cell.outputs:
        if out.output_type == "stream":
            print(out.text.rstrip())
        elif out.output_type == "execute_result":
            print(out.data["text/plain"])
```

```
--- cell 2 ---
np.float64(49.199999999999996)
--- cell 3 ---
Mean: 9.84
count      5.00
mean       9.84
```

```
std      9.93
min      0.00
25%      3.10
50%      8.20
75%     12.50
max     25.40
dtype: float64
```

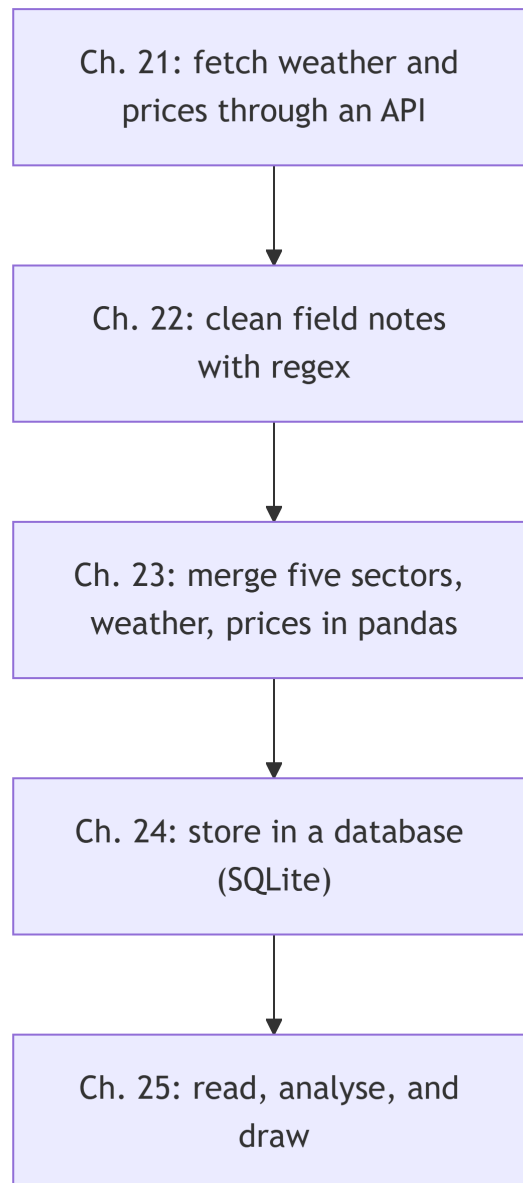
Cell 2 sums the rain: `np.float64(49.199999999999996)`. That is the real content of the cell result; computer decimals are not always exact (Chapter 5), and numpy 2 shows the type. Cell 3 prints the mean 9.84 and then the `describe()` result (the value of a cell's last expression is displayed automatically, without `print`). The notebook file that was made is at `_tmp/rain_notes.ipynb`; open it in JupyterLab to see how it looks.

TECHNICAL STUFF (feel free to skip)

A notebook keeps the **state** of the kernel (the order cells were run). Cells run out of order can give results that cannot be reproduced from scratch. A good habit: before sharing a notebook, choose “Restart kernel and run all” and make sure every cell still works. For reproducible reports, Chapter 29 covers Quarto and other ways.

Running Project I2: from database to chart

The close of Project I2. The script saves the fictional data into SQLite (a summary of Chapters 21-24), **reads it back** with one SQL query, analyses it with pandas, and draws two charts.



Running project I2 from start to end: five chapters, one flow of data.

```
# Chapter 25 - Running project I2, final step: from database to chart
from pathlib import Path
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import pandas as pd
from sqlalchemy import create_engine

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
db_file = tmp / "closing_i2.db"
db_file.unlink(missing_ok=True)
engine = create_engine(f"sqlite:/// {db_file.as_posix()}")
```

```

# Steps 1-4 (Chapters 21-24) in short: the fictional data is saved in a database
harvest = pd.DataFrame(
    [("2026-01-01", "Forestry", "teak", 1200.0), ("2026-01-01", "Plantation", "coffee",
245.5),
    ("2026-01-02", "Plantation", "cocoa", 198.5), ("2026-01-02", "Cropland", "rice",
3100.0),
    ("2026-01-03", "Cropland", "maize", 1850.0), ("2026-01-03", "Fisheries", "catfish",
432.4),
    ("2026-01-04", "Livestock", "cattle", 1151.5), ("2026-01-04", "Plantation",
"coffee", 310.0),
    ("2026-01-05", "Cropland", "rice", 2950.0), ("2026-01-05", "Plantation", "cocoa",
276.0)],
    columns=["date", "sector", "commodity", "yield_kg"])
dates = ["2026-01-01", "2026-01-02", "2026-01-03", "2026-01-04", "2026-01-05"]
weather = pd.DataFrame({"date": dates, "rain_mm": [12.5, 0.0, 8.2, 25.4, 3.1]})
price_lists = {"rice": [6200, 6250, 6180, 6300, 6270], "maize": [5100, 5150, 5200,
5120, 5180],
    "coffee": [58000, 58500, 59000, 57500, 58200], "cocoa": [62000, 61500,
62500, 63000, 62800]}
prices = pd.DataFrame([(d, c, p) for c, v in price_lists.items() for d, p in zip(dates,
v)],
    columns=["date", "commodity", "price_per_kg"])
harvest.to_sql("harvest", engine, index=False)
weather.to_sql("weather", engine, index=False)
prices.to_sql("prices", engine, index=False)

# Step 5 (this chapter): read back from the database, merge, analyse, draw
df = pd.read_sql_query("""
    SELECT h.date, h.sector, h.yield_kg, w.rain_mm, h.yield_kg * p.price_per_kg AS
income_rp
    FROM harvest h
    LEFT JOIN weather w ON w.date = h.date
    LEFT JOIN prices p ON p.date = h.date AND p.commodity = h.commodity
""", engine)
engine.dispose()

per_sector = df.groupby("sector")["yield_kg"].sum().sort_values(ascending=False)
per_day = df.groupby("date").agg(yield_kg=("yield_kg", "sum"), rain_mm=("rain_mm",
"first"))
print(per_sector)
print(per_day)
print("Correlation of rain and total daily yield (only 5 days, do not trust it):",
    round(per_day["rain_mm"].corr(per_day["yield_kg"]), 2))
print("Recorded income (Rp):", f"{df['income_rp'].sum():.0f}")

fig, (left, right) = plt.subplots(1, 2, figsize=(10, 4))
per_sector.plot.bar(ax=left, color="tab:green", rot=30)
left.set_title("Yield per sector (kg)")
right.bar(per_day.index.str[-2:], per_day["yield_kg"], color="tab:olive")
right2 = right.twinx()
right2.plot(per_day.index.str[-2:], per_day["rain_mm"], color="tab:blue", marker="o")
right.set_title("Daily yield (bars) and rain (line)")
fig.tight_layout()
fig.savefig(tmp / "project_i2.png", dpi=100)
print("Chart saved:", (tmp / "project_i2.png").exists())

```

sector	
Cropland	7900.0
Forestry	1200.0
Livestock	1151.5

```

Plantation    1030.0
Fisheries     432.4
Name: yield_kg, dtype: float64
            yield_kg  rain_mm
date
2026-01-01    1445.5     12.5
2026-01-02    3298.5      0.0
2026-01-03    2282.4      8.2
2026-01-04    1461.5     25.4
2026-01-05    3226.0      3.1
Correlation of rain and total daily yield (only 5 days, do not trust it): -0.87
Recorded income (Rp): 109,096,050
Chart saved: True

```

The result: Cropland is the largest (7900.0 kg), followed by Forestry (1200.0), Livestock (1151.5), Plantation (1030.0), and Fisheries (432.4). The recorded income of Rp 109,096,050 is the sum of Plantation and Cropland only (61,604,550 + 47,491,500), because only they have prices. The correlation between rain and the total daily yield is **-0.87**, which looks strong, but it comes from **only 5 days**; it is an interesting coincidence, **not** proof that rain lowers harvests. Measuring how sure we can be about numbers like that is the job of statistics (Series I3).

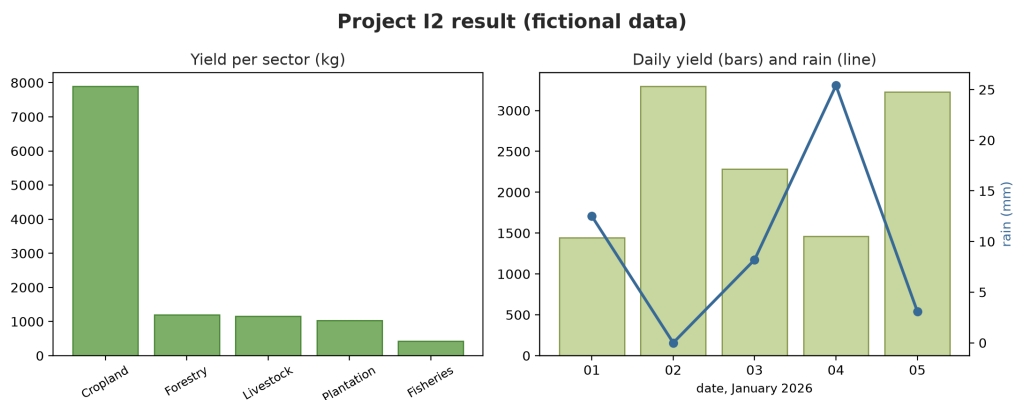


Figure 0.1: Project I2 result: yield per sector (left) and daily yield with rain (right). Fictional data.

Exercises

1. In `01_pandas_advanced.py`, change the rain category boundaries to 0-10 (dry), 10-20 (moderate), over 20 (heavy). How do the day counts per category change?
2. In `02_time_series.py`, replace `resample("W")` with `resample("D")` and then `rolling(window=3)`. How does it differ from the 7-day rolling?
3. In `06_sklearn_regression.py`, change `test_size` to 0.5. Does `R2` change? Why?
4. **Challenge:** in `07_sklearn_classification.py`, change `max_depth` to 1, then 6. Compare training and test accuracy. Which model is the most honest?

Chapter Summary

- Advanced pandas: derived columns, `pd.cut` for categories, `groupby().agg()`, a date index with `resample` and `rolling`.
- numpy: `percentile`, `corrcoef`, `where`, `cumsum`, and **broadcasting** without loops.
- matplotlib: figure and axes; `plot`, `bar`, `scatter`; give a title, axes, and **units**.
- scikit-learn: split into training/test data, `fit`, `predict`, `R2` or accuracy; **compare with a simple baseline**.
- Results from small samples are only estimates; correlation is not cause and effect.
- **Jupyter Notebook** combines text, code, and results; **Anaconda** brings Jupyter and data science libraries (not run here).

Closing Series I2

Congratulations! You have travelled the whole journey of data: **fetching** data through an API, **cleaning** it with regex, **processing** it in numpy and pandas, **storing** it in a database, and **drawing** and **guessing** with matplotlib and scikit-learn. The five sectors, the weather, and the prices now live in one workflow that you can repeat.

But tidy data soon raises a harder question: **what does it mean?** Is the difference between the average of two plots real or a coincidence? How sure are we that rain affects harvests? Series I3 answers with **statistics** and modelling.

Next, in Chapter 26: statistics with Python, from means and spread to hypothesis tests and how to interpret a p-value correctly.

Chapter 26 — Statistics with Python: From Numbers to Decisions

Series I3 — Statistics and Modelling

In this chapter, you will:

- Summarise data with descriptive statistics (mean, median, spread)
- Calculate probabilities with the normal, binomial, and Poisson distributions
- Build confidence intervals from a sample
- Run hypothesis tests: t-test, chi-square, ANOVA, and their nonparametric backups
- Read p-values correctly, interpret effect sizes, and tell correlation from cause and effect

All scripts for this chapter are in the `code/ch26/` folder. On the web version, you can also [download them all at once \(zip\)](#).

In Series I2, you fetched, cleaned, and stored data. Tidy data does not yet answer the real question: “*Do trees in plot B really grow faster, or is that just chance?*” Answering it takes **statistics**, an honest way of thinking about numbers that wobble. Imagine tasting one spoonful of soup to judge the whole pot: you do not need to finish the pot, as long as you stirred before scooping. That is the heart of this chapter: judging the big from the small, while knowing how confident we are allowed to feel.

i REMEMBER

All data in this chapter is **made up (fictional)**, produced by scripts with a fixed seed (`default_rng(42)`), so the numbers are the same every run. They are not real research results and must not be used as farming advice.

The libraries are the ones introduced in Series I2 (`numpy`, `pandas`, `matplotlib`) plus three new ones: **scipy** (`scipy.stats`: distributions and classic tests), **statsmodels** (models and report-style ANOVA tables), and **pingouin** (statistical tests with compact result tables, including effect sizes). For plots, we use **seaborn**.

Descriptive statistics: summarising without deceiving

You measured the diameter growth of 30 trees in plot A (cm per year). Staring at 30 numbers does not help, so we condense them.

```
# Chapter 26 - Descriptive statistics: summarising tree data
import numpy as np
import pandas as pd

rng = np.random.default_rng(42)
# Made-up (fictional) data: diameter growth (cm per year) of 30 trees in plot A
growth = rng.normal(loc=1.8, scale=0.6, size=30).round(2)
growth[0] = 4.9 # one extraordinarily fast tree (an outlier)

s = pd.Series(growth, name="growth_cm")
print("Number of values :", s.count())
print("Mean           :", round(s.mean(), 3))
print("Median          :", round(s.median(), 3))
print("Standard dev.    :", round(s.std(), 3))
print("Quartile 1 and 3 :", s.quantile(0.25).round(3), s.quantile(0.75).round(3))
print("Min and max       :", s.min(), s.max())
print("Skewness         :", round(s.skew(), 3))
```



```

without = s.drop(0)
print("Mean without outlier :", round(without.mean(), 3))
print("Median without outlier:", round(without.median(), 3))
print()
print(s.describe().round(3))

```

```

Number of values : 30
Mean             : 1.908
Median           : 1.86
Standard dev.    : 0.732
Quartile 1 and 3 : 1.552 2.218
Min and max      : 0.63 4.9
Skewness         : 2.198
Mean without outlier : 1.804
Median without outlier: 1.84

```

```

count    30.000
mean      1.908
std       0.732
min       0.630
25%       1.552
50%       1.860
75%       2.218
max       4.900
Name: growth_cm, dtype: float64

```

- **Mean** = the sum divided by the count. **Median** = the middle value when sorted. Both are “central numbers”, but they behave differently.
- **Standard deviation** = a rough measure of how far the data typically spreads from its mean. **Quartiles** cut the sorted data into four equal-sized parts.
- One “magic” tree worth 4.9 pulls the mean up to 1.908, while without that tree the mean is 1.804. The **median barely moves** (1.86 becomes 1.84). So when the data has outliers or is skewed (skewness 2.198, far from 0), the median is often more honest than the mean.
- `describe()` gives the five-number summary in one go.

TIP

Always **plot first**, then calculate. One histogram or *boxplot* reveals outliers and skew that a column of numbers hides. The seaborn section below shows how.

Probability and distributions

A **distribution** is a “map” of how likely each value is. `scipy.stats` offers many distributions with uniform methods: `cdf` (probability below a value), `sf` (probability above it), `pmf` (probability of exactly one value, for counts), and `ppf` (the inverse of `cdf`: the value for a given probability).

```

# Chapter 26 - Probability and distributions with scipy.stats
from scipy import stats

# Normal: tree diameter averages 30 cm, standard deviation 5 cm
d = stats.norm(loc=30, scale=5)
print("P(diameter < 25)      =", round(d.cdf(25), 4))
print("P(diameter > 40)      =", round(d.sf(40), 4))
print("P(25 < diameter < 35) =", round(d.cdf(35) - d.cdf(25), 4))
print("90th percentile diameter =", round(d.ppf(0.90), 2))

# Binomial: 20 seeds sown, each sprouts with probability 0.8

```

```

b = stats.binom(n=20, p=0.8)
print()
print("Average seeds sprouting =", b.mean())
print("P(exactly 16 sprout)   =", round(b.pmf(16), 4))
print("P(at least 18 sprout)  =", round(b.sf(17), 4))

# Poisson: an average of 3 fallen trees per hectare per season
p = stats.poisson(mu=3)
print()
print("P(no tree falls)      =", round(p.pmf(0), 4))
print("P(more than 5 fall)    =", round(p.sf(5), 4))

```

```

P(diameter < 25)          = 0.1587
P(diameter > 40)          = 0.0228
P(25 < diameter < 35)    = 0.6827
90th percentile diameter = 36.41

```

```

Average seeds sprouting = 16.0
P(exactly 16 sprout)    = 0.2182
P(at least 18 sprout)   = 0.2061

```

```

P(no tree falls)       = 0.0498
P(more than 5 fall)     = 0.0839

```

- **Normal** (bell curve): measurements often look like this. Tree diameter averaging 30 cm with a standard deviation of 5 cm: about 68% of trees (0.6827) lie within one standard deviation of the mean (25 to 35 cm), and only 2.28% exceed 40 cm.
- **Binomial**: the number of “successes” in several yes/no trials. 20 seeds with a sprouting chance of 0.8 yield 16 seedlings on average; the chance of exactly 16 is only 0.2182, not 1. The chance of at least 18 sprouting is about 21%.
- **Poisson**: the number of rare events in a region or time span (fallen trees per hectare). With an average of 3, the chance that none falls is only 4.98%.

Sampling and confidence intervals

We almost never measure **all** trees (the population). We take a **sample** and estimate. An estimate from a sample is bound to miss a little, and the **standard error** measures how much: the standard deviation divided by the square root of the count. A 95% **confidence interval** is a range around the estimate made with a recipe that, repeated many times, captures the true value about 95% of the time.

```

# Chapter 26 - Sampling and confidence intervals
import numpy as np
from scipy import stats

rng = np.random.default_rng(42)
# Fictional "population": heights of 5000 trees (m)
population = rng.normal(loc=18.0, scale=4.0, size=5000)
print("Population mean (the true value):", round(population.mean(), 3))

# Draw one random sample of 40 trees
sample = rng.choice(population, size=40, replace=False)
mean = sample.mean()
std_error = stats.sem(sample) # standard deviation / square root of n
print("Sample mean          :", round(mean, 3))
print("Standard error       :", round(std_error, 3))

# 95% confidence interval using the t distribution
low, high = stats.t.interval(0.95, df=len(sample) - 1, loc=mean, scale=std_error)
print("95% interval        :", round(low, 2), "to", round(high, 2))

```

```
# Repeat 1000 times: how often does the interval catch the true value?
caught = 0
for _ in range(1000):
    s = rng.choice(population, size=40, replace=False)
    lo, hi = stats.t.interval(0.95, df=39, loc=s.mean(), scale=stats.sem(s))
    caught += lo <= population.mean() <= hi
print("Intervals that caught the true value:", caught, "out of 1000")
```

```
Population mean (the true value): 17.92
Sample mean          : 17.714
Standard error       : 0.617
95% interval         : 16.47 to 18.96
Intervals that caught the true value: 961 out of 1000
```

The true value (population mean) is 17.92 m. Our sample estimates 17.714 with an interval of 16.47 to 18.96, which indeed captures 17.92. Then we repeat 1000 times: 961 intervals capture the true value, close to the promised 95%.

40 confidence intervals of 95%: about 5% miss the true value

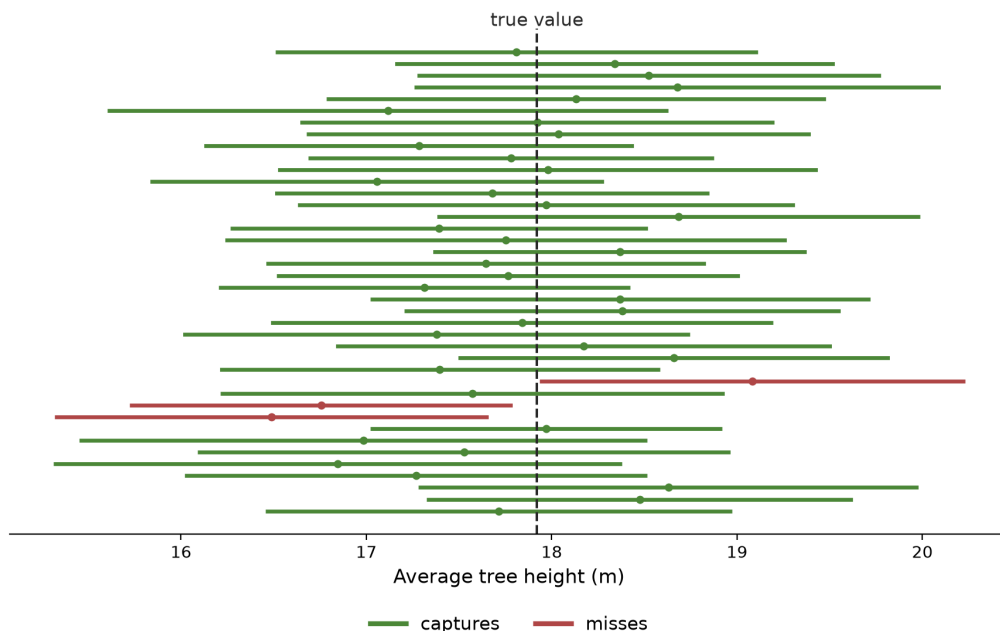


Figure 0.1: Each line is one 95% interval from one sample. About 5% (red) miss the true value; we cannot tell which ones while working in the field.

⚠ WARNING

A 95% interval does **not** mean “there is a 95% chance the true value lies inside this interval”. The true value is fixed (not random); what is random is the interval. The correct statement: this *recipe* captures the true value in 95% of the samples we might draw. More data makes the interval narrower.

Hypothesis tests: two groups

Now the opening question of this chapter. Plots A and B are compared with a **hypothesis test**:

1. **Null hypothesis** (H_0): there is no real difference; the visible gap is just chance.
2. Calculate how rare data like this would be if H_0 were true. The measure is the **p-value**.

3. If the p-value is small (by habit: below 0.05, the **significance level**), we reject H0.

```
# Chapter 26 - t-test: does plot A grow differently from plot B?
import numpy as np
import pingouin as pg
from scipy import stats

rng = np.random.default_rng(42)
plot_a = rng.normal(1.8, 0.6, 30).round(2) # cm per year (fictional)
plot_b = rng.normal(2.2, 0.7, 30).round(2)

print("Mean A:", plot_a.mean().round(3), " Mean B:", plot_b.mean().round(3))

# Welch t-test (does not assume equal variances)
result = stats.ttest_ind(plot_a, plot_b, equal_var=False)
print("t =", round(result.statistic, 3), " p =", round(result.pvalue, 4))

# pingouin gives a full table, including the effect size (Cohen d)
table = pg.ttest(plot_a, plot_b, correction=True)
print(table[["T", "dof", "p_val", "cohen_d", "CI95"]].round(4).to_string())

# Paired test: the same 12 trees measured before and after fertilising
before = rng.normal(1.5, 0.4, 12)
after = before + rng.normal(0.3, 0.25, 12)
paired = stats.ttest_rel(after, before)
print("Paired: t =", round(paired.statistic, 3), " p =", format(paired.pvalue, ".1e"))
```

```
Mean A: 1.81 Mean B: 2.28
t = -3.512 p = 0.0009
      T      dof    p_val   cohen_d      CI95
T_test -3.5121  56.012  0.0009   0.9068  [-0.74, -0.2]
Paired: t = 7.457 p = 1.3e-05
```

- Mean A is 1.81 and B is 2.28. The **Welch t-test** gives $t = -3.512$ and $p = 0.0009$. The minus sign is only because A is subtracted from B. Welch does not assume equal variances in the two plots, so it is the safer first choice.
- `pingouin.ttest` gives the same table plus **Cohen d** = 0.9068 and a confidence interval for the **difference** in means (-0.74 to -0.2). The interval does not contain 0, in line with the small p.
- The **paired test** (`ttest_rel`) is used when the same trees are measured twice (before and after fertilising). It is more sensitive because it compares each tree with itself.

i REMEMBER

A **p-value** is not the probability that your hypothesis is true. $p = 0.0009$ means: *if* there were no real difference, data this extreme would occur only about 9 times in 10,000 draws. It does not say how **big** the difference is or how **important** it is. That is the job of an effect size.

An **effect size** answers “how big”. Cohen d is the difference in means divided by the pooled standard deviation. Cohen’s rough guide: 0.2 small, 0.5 medium, 0.8 large. The 0.9 above counts as large.

Chi-square: two categories

The t-test is for numbers; for **counts per category** we use **chi-square**. Is tree species related to healthy/sick?

```
# Chapter 26 - Chi-square: is tree species related to health?
import pandas as pd
from scipy import stats
```

```
# Fictional contingency table (number of trees)
table = pd.DataFrame(
    {"healthy": [42, 35, 28], "sick": [8, 15, 22]},
    index=["Teak", "Meranti", "Mahogany"],
)
print(table)

chi2, p, dof, expected = stats.chi2_contingency(table)
print()
print("chi2 =", round(chi2, 3), " degrees of freedom =", dof, " p =", round(p, 4))
print("Expected values if there were no relationship:")
print(pd.DataFrame(expected, index=table.index, columns=table.columns).round(1))

# Effect size: Cramer V
n = table.values.sum()
v = (chi2 / (n * (min(table.shape) - 1))) ** 0.5
print("Cramer V =", round(v, 3))
```

	healthy	sick
Teak	42	8
Meranti	35	15
Mahogany	28	22

```
chi2 = 9.333 degrees of freedom = 2 p = 0.0094
Expected values if there were no relationship:
      healthy sick
Teak      35.0  15.0
Meranti    35.0  15.0
Mahogany   35.0  15.0
Cramer V = 0.249
```

If species and health were unrelated, each species (50 trees each) would be expected to have 35 healthy and 15 sick. What was observed deviates: Teak is healthier (42), Mahogany sicker (22). Chi-square = 9.333 with 2 degrees of freedom gives $p = 0.0094$. The effect size **Cramer V** = 0.249, a real relationship but not a very strong one.

ANOVA: three or more groups

Comparing three varieties with three pairwise t-tests raises the risk of error (see the traps below). **ANOVA** tests all groups at once: “is at least one different?” It assumes each group’s data is roughly normal and the variances are similar, so we first check with the Shapiro and Levene tests.

```
# Chapter 26 - ANOVA for three varieties, and a nonparametric backup
import numpy as np
import pandas as pd
import pingouin as pg
from scipy import stats

rng = np.random.default_rng(42)
data = pd.DataFrame({
    "variety": np.repeat(["V1", "V2", "V3"], 25),
    "height_cm": np.concatenate([
        rng.normal(120, 12, 25), rng.normal(128, 12, 25), rng.normal(137, 12, 25)
    ]).round(1),
})
print(data.groupby("variety")["height_cm"].agg(["mean", "std"]).round(2))

groups = [g["height_cm"].values for _, g in data.groupby("variety")]
print()
print("Shapiro (normal?) p per group:", [round(float(stats.shapiro(k).pvalue), 3) for
```

```

k in groups])
print("Levene (equal variances?) p :", round(stats.levene(*groups).pvalue, 3))

anova = pg.anova(data=data, dv="height_cm", between="variety", detailed=False)
print()
print("F =", round(anova.loc[0, "F"], 2), " p =", format(anova.loc[0, "p_unc"], ".2e"),
      " partial eta-squared =", round(anova.loc[0, "np2"], 3))

print()
print(pg.pairwise_tukey(data=data, dv="height_cm", between="variety")[["A", "B",
"diff", "p_tukey"]].round(4).to_string())

# Nonparametric backup (does not assume a normal distribution)
print()
print("Kruskal-Wallis p =", format(stats.kruskal(*groups).pvalue, ".2e"))
print("Mann-Whitney V1 vs V3 p =", round(stats.mannwhitneyu(groups[0],
groups[2]).pvalue, 5))

```

```

          mean  std
variety
V1      119.58  9.97
V2      130.61  8.32
V3      135.45  9.94

```

```

Shapiro (normal?) p per group: [0.572, 0.168, 0.851]
Levene (equal variances?) p : 0.433

```

```

F = 18.55  p = 3.17e-07  partial eta-squared = 0.34

```

```

      A  B  diff  p_tukey
0  V1 V2 -11.036  0.0003
1  V1 V3 -15.872  0.0000
2  V2 V3  -4.836  0.1734

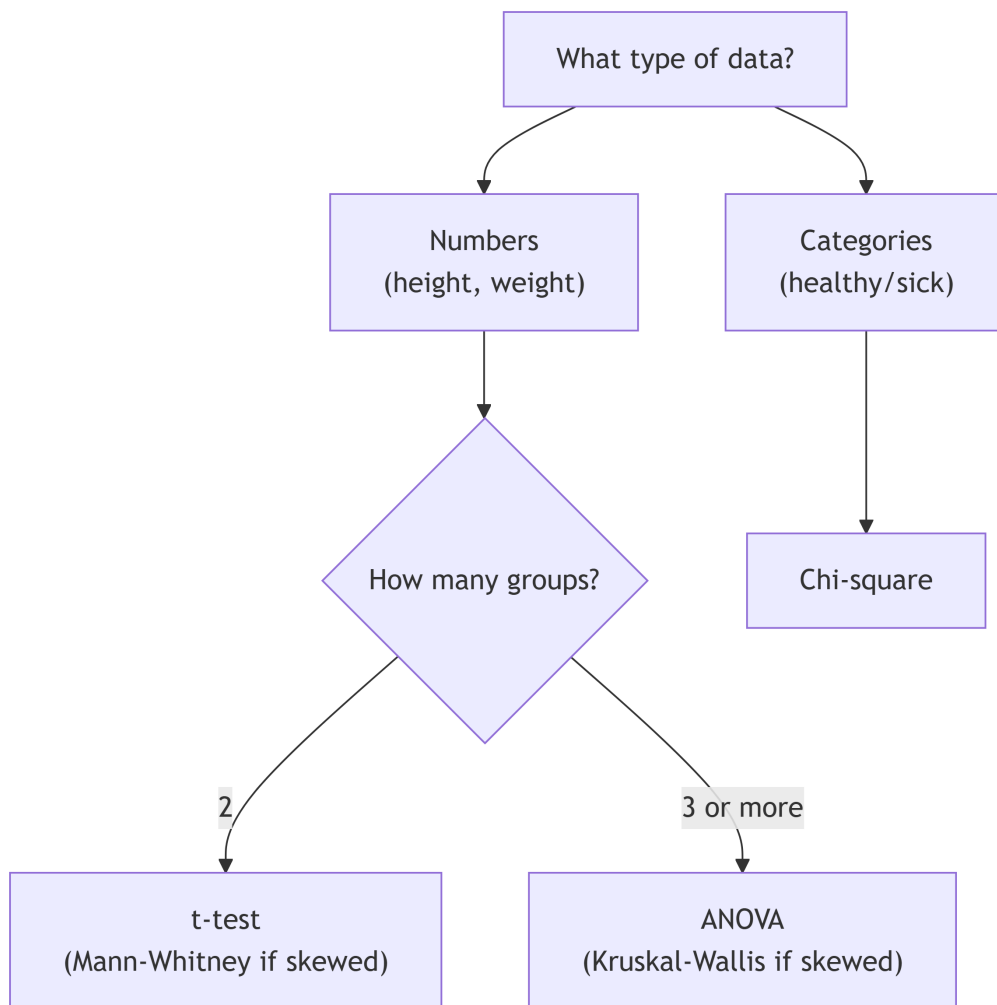
```

```

Kruskal-Wallis p = 4.19e-06
Mann-Whitney V1 vs V3 p = 1e-05

```

- Mean height of V1, V2, V3: 119.58, 130.61, 135.45 cm. The Shapiro test (all p above 0.05) and Levene (p = 0.433) show no violation of the assumptions.
- ANOVA: F = 18.55, p = 3.17e-07 (meaning 0.000000317). Partial eta-squared 0.34: about 34% of the variation in height is associated with variety.
- ANOVA only says “something differs”. The **Tukey follow-up test** shows which: V1-V2 (p = 0.0003) and V1-V3 (0.0000, meaning below 0.00005) differ, while V2-V3 (0.1734) is not yet shown to differ.
- A **nonparametric test** does not assume a normal distribution; it works on the ranks of the data. Kruskal-Wallis replaces ANOVA, Mann-Whitney replaces the t-test. Here both agree with ANOVA. Use them when data is very skewed, consists of ranks, or the sample is tiny.



Choosing a hypothesis test: start from the type of data, then the number of groups.

Two p-value traps

```
# Chapter 26 - Two p-value traps: many tests and giant samples
import numpy as np
from scipy import stats

rng = np.random.default_rng(42)

# Trap 1: 20 tests on data from the same population (no real difference)
significant = 0
for i in range(20):
    a = rng.normal(0, 1, 30)
    b = rng.normal(0, 1, 30)
    p = stats.ttest_ind(a, b).pvalue
    if p < 0.05:
        significant += 1
        print(f"  test {i + 1}: p = {p:.4f}  <- 'significant' although there is
no difference")
print("'Significant' tests out of 20:", significant)
```

```

print("After Bonferroni correction, the threshold becomes", 0.05 / 20)

# Trap 2: a tiny difference, a gigantic sample
a = rng.normal(100.0, 15, 200_000)
b = rng.normal(100.3, 15, 200_000)
result = stats.ttest_ind(a, b)
d = (b.mean() - a.mean()) / 15
print()
print("Mean difference:", round(b.mean() - a.mean(), 3), " p =", format(result.pvalue, ".1e"))
print("Effect size (d) :", round(d, 3), "-> very small, even though p is tiny")

```

```

test 15: p = 0.0327 <- 'significant' although there is no difference
'Significant' tests out of 20: 1
After Bonferroni correction, the threshold becomes 0.0025

Mean difference: 0.323 p = 9.7e-12
Effect size (d) : 0.022 -> very small, even though p is tiny

```

- **Trap 1, too many tests.** We ran 20 tests on data that actually comes from the same population. One test (the 15th, $p = 0.0327$) looks “significant”. With a threshold of 0.05, about one in 20 tests of no real difference *will* look significant. If you test many things and report only the “significant” ones, you are harvesting chance. The cure: state hypotheses beforehand, and use a **correction** such as Bonferroni (threshold divided by the number of tests, here 0.0025).
- **Trap 2, a giant sample.** With 200,000 values per group, a difference of 0.323 points on a scale with a standard deviation of 15 ($d = 0.022$) yields $p = 9.7e-12$. Statistically significant, yet barely meaningful in the real world. **Significant does not mean important.** Always report the effect size and confidence interval next to the p-value.

Correlation is not causation

Correlation measures how tightly two numbers rise and fall together (from -1 to 1). Pearson is for straight-line relationships; Spearman uses ranks, so it resists outliers.

```

# Chapter 26 - Correlation is not causation: a hidden factor
import numpy as np
import pandas as pd
from scipy import stats
import pingouin as pg

rng = np.random.default_rng(42)
n = 60
rain = rng.normal(200, 50, n)          # rainfall (mm), the hidden factor
fungus = 0.08 * rain + rng.normal(0, 3, n) # fungus grows more when it rains
harvest = 0.5 * rain + rng.normal(0, 25, n) # the harvest also rises with rain
df = pd.DataFrame({"rain_mm": rain, "fungus": fungus, "harvest_kg": harvest})

r, p = stats.pearsonr(df["fungus"], df["harvest_kg"])
print("Correlation fungus vs harvest (Pearson):", round(r, 3), " p =", format(p, ".1e"))
rho, p2 = stats.spearmanr(df["fungus"], df["harvest_kg"])
print("Rank correlation (Spearman)          :", round(rho, 3))

# Partial correlation: remove the influence of rain
partial = pg.partial_corr(data=df, x="fungus", y="harvest_kg", covar="rain_mm")
print()
print("Correlation fungus vs harvest once rain is controlled:")
print(partial[["n", "r", "p_val"]].round(3).to_string())

```



```
Correlation fungus vs harvest (Pearson): 0.513 p = 2.8e-05
Rank correlation (Spearman) : 0.504
```

Correlation fungus vs harvest once rain is controlled:

```
      n      r  p_val
pearson 60 0.057 0.667
```

Fungus and harvest correlate at 0.513 ($p = 2.8e-05$): the relationship looks strong. Does wiping out fungus lower the harvest? No. In this data, **rain** raises both (fungus loves damp, crops love water). After controlling for rain with a **partial correlation**, the fungus-harvest relationship drops to 0.057 with $p = 0.667$, in other words it vanishes. Rain is here called a **confounder**.

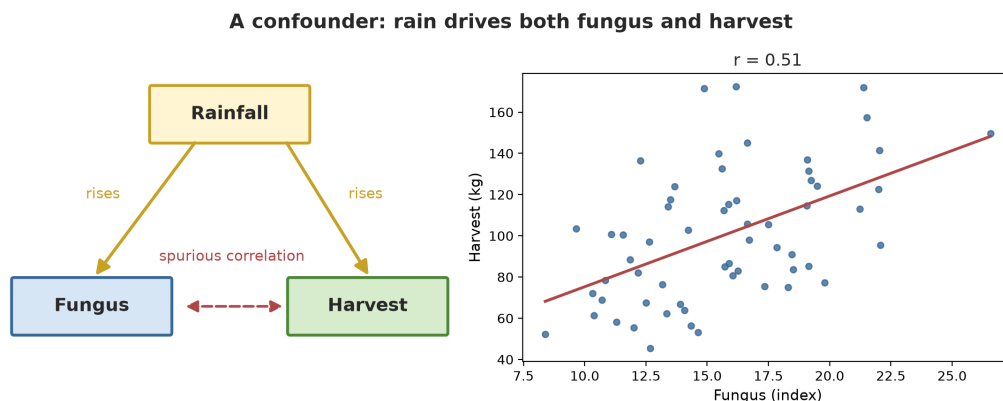


Figure 0.2: A confounder: rain drives both fungus and harvest, so the two seem correlated.

⚠ WARNING

Cause and effect can only be concluded from a **controlled experiment** (for example plots given treatments at random) or from very careful analysis. Ordinary observational data only shows relationships. Beware of the sentence “X causes Y” built on correlation alone.

Statistical visualisation with seaborn

```
# Chapter 26 - Statistical visualisation with seaborn
from pathlib import Path

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

rng = np.random.default_rng(42)
data = pd.DataFrame({
    "plot": np.repeat(["A", "B", "C"], 30),
    "growth_cm": np.concatenate([
        rng.normal(1.8, 0.6, 30), rng.normal(2.3, 0.7, 30), rng.normal(2.0, 0.5, 30)
    ]),
})

out = Path(__file__).parent / "_tmp"
out.mkdir(exist_ok=True)
```

```
fig, axes = plt.subplots(1, 2, figsize=(9, 3.6))
sns.boxplot(data=data, x="plot", y="growth_cm", ax=axes[0])
sns.stripplot(data=data, x="plot", y="growth_cm", color="black", size=3, ax=axes[0])
sns.histplot(data=data, x="growth_cm", hue="plot", kde=True, ax=axes[1])
fig.tight_layout()
fig.savefig(out / "seaborn_plot.png", dpi=120)
print("Figure saved:", (out / "seaborn_plot.png").name)
print("Size per plot:", data.groupby("plot").size().to_dict())
```

```
Figure saved: seaborn_plot.png
Size per plot: {'A': 30, 'B': 30, 'C': 30}
```

The script saves the figure into the `_tmp/` folder (not shown in the book). Open and look: the *boxplot* shows the median, quartiles, and outliers; the dots (*stripplot*) show the raw data; the histogram with a *kde* line shows the shape of the spread. Combining the three means a single average rarely fools you.

Running Project I3: growth differences between plots and varieties

The first task of project I3: tree diameter growth (cm per year) in three plots (A, B, C) and two varieties (V1, V2), 20 trees per combination. All numbers are fictional, with differences we plant ourselves (B higher by 0.5; V2 higher by 0.3), so we can check whether the tests find them.

```
# Chapter 26 - Running project I3: growth differences between plots and varieties
import numpy as np
import pandas as pd
import pingouin as pg
import statsmodels.api as sm
import statsmodels.formula.api as smf

rng = np.random.default_rng(42)
plot_effect = {"A": 0.0, "B": 0.5, "C": 0.2}      # fictional
variety_effect = {"V1": 0.0, "V2": 0.3}          # fictional
rows = []
for p, ep in plot_effect.items():
    for v, ev in variety_effect.items():
        for g in rng.normal(1.8 + ep + ev, 0.6, 20):
            rows.append({"plot": p, "variety": v, "growth_cm": round(g, 2)})
trees = pd.DataFrame(rows)

print(trees.pivot_table(index="plot",          columns="variety",          values="growth_cm",
aggfunc="mean").round(2))

model = smf.ols("growth_cm ~ C(plot) + C(variety)", data=trees).fit()
anova = sm.stats.anova_lm(model, typ=2)
print()
print(anova.round(4))

print()
print("Differences between plots (Tukey):")
print(pg.pairwise_tukey(data=trees, dv="growth_cm", between="plot")["A", "B", "diff",
"p_tukey", "hedges"]).round(3).to_string())

v1 = trees.query("variety == 'V1'")["growth_cm"]
v2 = trees.query("variety == 'V2'")["growth_cm"]
test = pg.ttest(v1, v2)
print()
```

```
print("V1 vs V2: p =", test["p_val"].round(4).iloc[0], " Cohen d =",
test["cohen_d"].round(2).iloc[0], " CI95 of difference =", test["CI95"].iloc[0])
```

```
variety    V1    V2
plot
A         1.78  2.17
B         2.37  2.54
C         1.79  2.23
```

```
              sum_sq      df      F  PR(>F)
C(plot)      5.8334      2.0  13.1621  0.0000
C(variety)    3.3534      1.0  15.1327  0.0002
Residual     25.7053     116.0      NaN      NaN
```

Differences between plots (Tukey):

```
   A B  diff  p_tukey  hedges
0  A B -0.485    0.000  -0.984
1  A C -0.037    0.942  -0.069
2  B C  0.448    0.000   0.922
```

```
V1 vs V2: p = 0.0006  Cohen d = 0.65  CI95 of difference = [-0.52 -0.15]
```

- The first table shows the mean of each combination. Plot B is highest in both varieties, and V2 is always above V1.
- `ols("growth_cm ~ C(plot) + C(variety)")` builds a model with two factors at once; `anova_lm` summarises it. Plot: $F = 13.16$, variety: $F = 15.13$, both p below 0.001 (displayed as 0.0000 and 0.0002 because of rounding).
- Tukey between plots: A-B differ (difference -0.485 ; Hedges effect size -0.984), B-C differ, but **A-C do not** ($p = 0.942$, difference -0.037). The tests found the differences we planted, and did not invent one for the pair that truly is similar.
- V1 vs V2: $p = 0.0006$, Cohen $d = 0.65$ (medium), the difference in means lies in the range -0.52 to -0.15 cm per year.

This project continues in Chapter 27 with regression, and all results go into the Quarto report in Chapter 29.

Exercises

1. Change `01_descriptive.py`: replace the outlier 4.9 with 9.9. Which shifts more, the mean or the median? Why?
2. Using the normal distribution in `02_distributions.py`, calculate the probability of a tree diameter between 20 and 30 cm, and the diameter that exceeds 95% of trees.
3. In `03_sampling_ci.py`, change the sample size from 40 to 160. How does the width of the 95% interval change? (Hint: the standard error roughly halves.)
4. **Challenge:** create data for two plots with exactly the same mean, repeat the t-test 1000 times with different seeds, and count what percentage of tests give p below 0.05. Is it close to 5%? What does it mean?

Chapter Summary

- **Descriptive:** mean, median, standard deviation, quartiles. Outliers disturb the mean, not the median.
- **Distributions** in `scipy.stats`: `cdf`, `sf`, `pmf`, `ppf`; normal for measurements, binomial for success/failure, Poisson for rare events.
- **Confidence interval** 95% = a recipe that captures the true value in about 95% of samples; not a probability for one interval.
- **Hypothesis tests:** t-test (two groups), chi-square (categories), ANOVA (three or more, followed by Tukey), and nonparametric backups.
- A **p-value** is not the probability the hypothesis is true and does not measure effect size. Beware of many tests and giant samples; always report the **effect size**.

- **Correlation is not causation:** look for confounders and control for them.

Next, in Chapter 27: regression, the way to build models that explain and predict: trees, fish, livestock, harvests, and rain.

Chapter 27 — Regression and Quantitative Modelling: Building Models from Data

Series I3 — Statistics and Modelling

In this chapter, you will:

- Build simple and multiple regression models with statsmodels, then check their assumptions
- Use logistic regression (yes/no), Poisson GLM (counts), and Ridge/Lasso regularisation
- Fit non-linear curves, mixed-effects models, and time series forecasts (ARIMA)
- Meet spatial regression (GWR) and know what is and is not tested in this book
- Simulate uncertainty (Monte Carlo), optimise, and run a sensitivity analysis on a system dynamics model

All scripts for this chapter are in the `code/ch27/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Chapter 26 answered “do two groups differ?”. This chapter answers a more useful question: “*How much does X affect Y, and what is the forecast of Y if X changes?*” That is **regression**: drawing the best line (or curve) through a cloud of points, then using it to explain and predict. Imagine weighing cattle every month and recording their feed; regression concludes “each extra 1 kg of daily feed adds on average this many kg of weight”, complete with how sure you can be of that number.

i REMEMBER

All data in this chapter is **made up (fictional)**, produced by scripts with a fixed seed, and deliberately built from rules we know so we can check whether the model finds them again. Do not use the numbers as farming, price, or feed advice.

Simple linear regression: a tree allometric equation

Weighing the biomass of a tree means felling it. Foresters prefer to measure the **diameter** (easy) and predict biomass with an **allometric equation**. The relationship curves (a tree twice as thick is far more than twice as heavy), but if both sides are put through a *log*, it becomes a straight line: $\ln(\text{biomass}) = a + b \times \ln(\text{diameter})$.

```
# Chapter 27 - Simple linear regression: a tree allometric equation
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

rng = np.random.default_rng(42)
n = 60
diameter = rng.uniform(10, 60, n) # cm
biomass = 0.12 * diameter ** 2.4 * np.exp(rng.normal(0, 0.15, n)) # kg (fictional)
trees = pd.DataFrame({"diameter_cm": diameter, "biomass_kg": biomass})
trees["ln_d"] = np.log(trees["diameter_cm"])
trees["ln_b"] = np.log(trees["biomass_kg"])

# The true relationship curves; after taking logs it becomes a straight line
model = smf.ols("ln_b ~ ln_d", data=trees).fit()
print(model.params.round(3))
print("R squared      :", round(model.rsquared, 3))
```

```

print("95% interval ln_d:", model.conf_int().loc["ln_d"].round(3).tolist())
print("p-value ln_d      :", format(model.pvalues["ln_d"], ".1e"))

# Predict the biomass of a 40 cm tree (transform back from the log scale)
a, b = model.params["Intercept"], model.params["ln_d"]
correction = np.exp(model.mse_resid / 2)      # bias correction after the log
print()
print("Prediction without correction:", round(np.exp(a) * 40 ** b, 1), "kg")
print("Prediction with correction   :", round(np.exp(a) * 40 ** b * correction, 1), "kg")

# Compare with a straight line fitted directly on the original scale
straight = smf.ols("biomass_kg ~ diameter_cm", data=trees).fit()
print("R squared of the direct straight line:", round(straight.rsquared, 3))
print("Straight-line prediction at 5 cm      :",
      round(straight.predict(pd.DataFrame({"diameter_cm": [5]}))[0], 1), "kg (nonsense: it
      can go negative)")

```

```

Intercept    -2.140
ln_d         2.397
dtype: float64
R squared      : 0.99
95% interval ln_d: [2.333, 2.461]
p-value ln_d   : 1.9e-59

Prediction without correction: 813.5 kg
Prediction with correction   : 818.8 kg
R squared of the direct straight line: 0.908
Straight-line prediction at 5 cm      : -501.4 kg (nonsense: it can go negative)

```

- The formula " $\ln_b \sim \ln_d$ " reads " $\ln_b$ explained by $\ln_d$ ". `smf.ols(...).fit()` finds the line that minimises the sum of squared differences (the **least squares** method, *ordinary least squares*, OLS).
- The slope $b = 2.397$ with a 95% interval of 2.333 to 2.461. We planted 2.4 in the data, and the model found it again. When diameter rises by 1%, biomass rises by about 2.4%.
- **R squared** = 0.99: about 99% of the variation in $\ln(\text{biomass})$ is explained by $\ln(\text{diameter})$. A high R squared does not guarantee a correct model; it only shows the fit to this data.
- Predicting back to kilograms needs `np.exp`, and because the mean of logs is not the log of the mean, there is a small **bias correction**: 813.5 kg without correction, 818.8 kg with it (factor `exp(residual variance / 2)`).
- A straight line fitted directly on the original scale reaches only an R squared of 0.908 and predicts -501.4 kg for a 5 cm tree: **negative mass**. The choice of model shape matters.

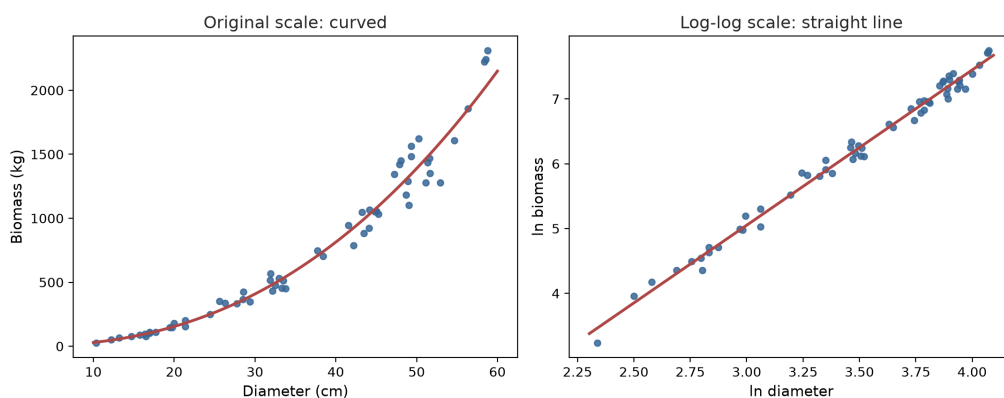


Figure 0.1: The allometric equation: curved on the original scale (left), straight after taking logs (right).

Multiple regression and diagnostics

Fish weight depends on many things at once. **Multiple regression** puts several predictors in one model; each coefficient means “the effect of that variable when the others are **held fixed**”.

```
# Chapter 27 - Multiple regression and diagnostics: fish growth versus feed
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
from scipy import stats
from statsmodels.stats.diagnostic import het_breuschpagan
from statsmodels.stats.outliers_influence import variance_inflation_factor

rng = np.random.default_rng(42)
n = 80
feed = rng.uniform(20, 60, n)          # grams of feed per fish per day
temp = rng.normal(28, 1.5, n)          # degrees Celsius
density = rng.uniform(10, 40, n)       # fish per m3
weight = 50 + 6.0 * feed + 12 * (temp - 28) - 2.5 * density + rng.normal(0, 25, n)
fish = pd.DataFrame({"weight_g": weight, "feed_g": feed, "temp_c": temp, "density":
density})

model = smf.ols("weight_g ~ feed_g + temp_c + density", data=fish).fit()
print(model.summary().tables[1])
print("R squared:", round(model.rsquared, 3), " adjusted R squared:",
round(model.rsquared_adj, 3))

# Diagnostic 1: are the residuals normally distributed?
print()
print("Shapiro residuals p  :", round(stats.shapiro(model.resid).pvalue, 3))
# Diagnostic 2: is the residual variance the same at every level? (homoskedasticity)
print("Breusch-Pagan p      :", round(het_breuschpagan(model.resid, model.model.exog)
[1], 3))

# Diagnostic 3: multicollinearity (VIF). Add a twin feed that is almost the same
fish["pellet_feed_g"] = 0.9 * fish["feed_g"] + rng.normal(0, 1.5, n)
X = sm.add_constant(fish[["feed_g", "temp_c", "density", "pellet_feed_g"]])
print()
for i, name in enumerate(X.columns):
    if name != "const":
        print(f"VIF {name:l4}: {variance_inflation_factor(X.values, i):6.1f}")
```

```
=====
              coef    std err          t      P>|t|      [0.025      0.975]
-----
Intercept    -336.0406     61.612     -5.454     0.000    -458.751    -213.330
feed_g         5.9493      0.282     21.133     0.000         5.389         6.510
temp_c        13.8181      2.183      6.330     0.000         9.470        18.166
density       -2.4164      0.325     -7.436     0.000        -3.064        -1.769
=====
```

R squared: 0.888 adjusted R squared: 0.883

Shapiro residuals p : 0.804
Breusch-Pagan p : 0.746

VIF feed_g : 43.4
VIF temp_c : 1.0
VIF density : 1.0
VIF pellet_feed_g : 43.4

- The coefficient table: each gram of feed adds 5.95 g of weight; each degree of temperature adds 13.8 g; each extra fish per m3 subtracts 2.42 g. The values we planted: 6, 12, and -2.5. The confidence interval (the last two columns) and p-value ($P > |t|$) are given for every row. 0.000 means below 0.0005.
- **Adjusted R squared** (0.883) penalises adding useless variables; use it to compare models with different numbers of variables.
- **Diagnostics** check whether the regression assumptions are reasonable. (1) The **residuals** (the differences between data and model) should be roughly normal: Shapiro $p = 0.804$, no sign of violation. (2) The residual variance should be the same at all levels (**homoskedasticity**): Breusch-Pagan $p = 0.746$, fine. (3) The predictors should not be too much alike (**multicollinearity**).
- **VIF** (*variance inflation factor*) measures that. We add `pellet_feed_g`, an almost-twin of `feed_g`, and the VIF of both jumps to 43.4 (rough rule: above 5 to 10 is worrying), while temperature and density stay at 1.0. As a result the coefficients of each twin become unstable. The cure: drop one, combine them, or use regularisation (next section).

TIP

Always **plot the residuals** against the predicted values. A curved pattern means the model shape is wrong; a funnel pattern (widening) means unequal variance. Diagnostics are not a formality: a model whose assumptions are broken gives misleading confidence intervals.

Logistic regression: a yes-or-no outcome

When only two outcomes are predicted (tree alive or dead, animal sick or healthy), a straight line would predict “probabilities” below 0 or above 1. **Logistic regression** predicts a **probability** (always between 0 and 1) through an S-shaped curve.

```
# Chapter 27 - Logistic regression: the probability a tree survives
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

rng = np.random.default_rng(42)
n = 200
diameter = rng.uniform(5, 50, n)
plot = rng.choice(["A", "B", "C"], n)
logit = -2.5 + 0.12 * diameter + np.where(plot == "C", -0.8, 0)
alive = rng.random(n) < 1 / (1 + np.exp(-logit))
trees = pd.DataFrame({"alive": alive.astype(int), "diameter_cm": diameter, "plot": plot})

model = smf.logit("alive ~ diameter_cm + C(plot)", data=trees).fit(dis= False)
print(model.params.round(3))

# Coefficients are on the log-odds scale; turn them into odds ratios
print()
print("Odds ratios (OR):")
print(np.exp(model.params).round(3))
print("Each extra 1 cm of diameter multiplies the survival odds by",
      round(np.exp(model.params["diameter_cm"]), 3))

# Probability for new trees
new = pd.DataFrame({"diameter_cm": [10, 30], "plot": ["A", "A"]})
print()
print("Survival probability (diameter 10 and 30 cm, plot A):",
      model.predict(new).round(3).tolist())

# Accuracy with a threshold of 0.5
predicted = (model.predict(trees) >= 0.5).astype(int)
```

```
print("Accuracy:", round((predicted == trees["alive"]).mean(), 3))
print("True proportion alive:", round(trees["alive"].mean(), 3))
```

```
Intercept      -2.385
C(plot)[T.B]   -0.084
C(plot)[T.C]   -0.487
diameter_cm     0.107
dtype: float64
```

Odds ratios (OR):

```
Intercept      0.092
C(plot)[T.B]    0.919
C(plot)[T.C]    0.614
diameter_cm     1.113
dtype: float64
```

Each extra 1 cm of diameter multiplies the survival odds by 1.113

Survival probability (diameter 10 and 30 cm, plot A): [0.211, 0.693]

Accuracy: 0.77

True proportion alive: 0.555

- The coefficients are on the **log-odds** scale, hard to read directly. `np.exp(coefficient)` turns them into **odds ratios** (OR): each extra 1 cm of diameter multiplies the survival odds by 1.113 (up about 11%). Plot C has an OR of 0.614: lower survival odds, as we planted.
- **Odds** = probability / (1 - probability). A probability of 0.5 equals odds of 1. Odds and probability are not the same thing, so do not say “11% more likely” for a probability.
- `predict` gives probabilities: a 10 cm tree in plot A about 0.211 and a 30 cm tree about 0.693.
- With a threshold of 0.5, accuracy is 0.77, better than guessing “all alive” (true proportion alive 0.555). For imbalanced cases (a rare disease), accuracy alone misleads; check precision and recall too, in Series A3.

GLM: count data with Poisson

The number of fruits per tree, the number of eggs, or the number of fallen trees are non-negative whole numbers. A **GLM** (*generalized linear model*) extends regression to other families of distributions. For counts, the **Poisson** family with a log link is used.

```
# Chapter 27 - Poisson GLM for count data: number of fruits per tree
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

rng = np.random.default_rng(42)
n = 150
diameter = rng.uniform(10, 50, n)
mu = np.exp(0.5 + 0.06 * diameter)
mean = mu * rng.gamma(shape=4, scale=0.25, size=n) # extra variation between trees
fruits = rng.poisson(mean)
data = pd.DataFrame({"fruits": fruits, "diameter_cm": diameter})

poisson = smf.glm("fruits ~ diameter_cm", data=data,
family=sm.families.Poisson()).fit()
print(poisson.params.round(4))
print("Rate ratio (IRR) per cm:", round(np.exp(poisson.params["diameter_cm"]), 4))

# Check overdispersion: Pearson chi-square divided by degrees of freedom; ideally
about 1
dispersion = poisson.pearson_chi2 / poisson.df_resid
print("Dispersion:", round(dispersion, 2), "(far above 1 means Poisson is too
```

```
optimistic)")

# Alternative: negative binomial, which estimates the extra variation (alpha) itself
nb = smf.negativebinomial("fruits ~ diameter_cm", data=data).fit(dis=False)
print()
print("Negative binomial, alpha :", round(nb.params["alpha"], 3))
print("SE diameter Poisson      :", round(poisson.bse["diameter_cm"], 5))
print("SE diameter neg. binomial:", round(nb.bse["diameter_cm"], 5))
```

```
Intercept      0.6400
diameter_cm    0.0577
dtype: float64
Rate ratio (IRR) per cm: 1.0594
Dispersion: 4.64 (far above 1 means Poisson is too optimistic)
```

```
Negative binomial, alpha : 0.267
SE diameter Poisson      : 0.00234
SE diameter neg. binomial: 0.00477
```

- $\exp(0.0577) = 1.0594$: each extra 1 cm of diameter raises the average number of fruits by about 5.9% (the **rate ratio**, IRR).
- Poisson assumes **variance = mean**. Real data is often more variable (**overdispersion**). The measure: Pearson chi-square divided by the degrees of freedom. Here it is **4.64**, far above 1. As a result the Poisson standard errors are too small and the conclusions too confident.
- The **negative binomial** estimates the extra variation itself ($\alpha = 0.267$). The standard error of diameter becomes 0.00477, almost double the 0.00234 of Poisson. That is the price of honesty.

Ridge and Lasso: when predictors are many and alike

With many mutually related predictors, OLS estimates wild coefficients (positive in one place, negative in another) and **overfits**: good on training data, poor on new data. **Regularisation** adds a “penalty” for large coefficients. **Ridge** shrinks all coefficients; **Lasso** shrinks and can set some to zero, selecting variables at the same time.

```
# Chapter 27 - Regularisation (Ridge and Lasso): cattle weight from many body
measurements
import numpy as np
from sklearn.linear_model import LassoCV, LinearRegression, RidgeCV
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

rng = np.random.default_rng(42)
n, p = 90, 10
base = rng.normal(size=(n, 1))
# 10 closely related measurements; only 3 truly matter
X = base + rng.normal(0, 0.5, size=(n, p))
true_coef = np.array([30, 20, 10, 0, 0, 0, 0, 0, 0, 0])
weight = 350 + X @ true_coef / 3 + rng.normal(0, 15, n) # kg (fictional)

X_train, X_test, y_train, y_test = train_test_split(X, weight, test_size=0.3,
random_state=42)

models = {
    "Plain OLS": make_pipeline(StandardScaler(), LinearRegression()),
    "Ridge": make_pipeline(StandardScaler(), RidgeCV(alphas=np.logspace(-2, 3, 30))),
    "Lasso": make_pipeline(StandardScaler(), LassoCV(cv=5, random_state=42)),
}
for name, m in models.items():
```

```

m.fit(X_train, y_train)
coef = m[-1].coef_
print(f"{name:10} test R2 = {m.score(X_test, y_test):.3f}    zero coefficients:
{int((np.abs(coef) < 1e-9).sum())} of {p}")
print("          ", np.round(coef, 1) + 0.0)

```

```

Plain OLS test R2 = 0.680    zero coefficients: 0 of 10
          [12.9  0.5 -4.8  6.5 -1.8 -0.4  3.6  2.   2.5 -0.4]
Ridge     test R2 = 0.754    zero coefficients: 0 of 10
          [ 8.5  2.2 -2.   4.3 -1.1  0.6  2.5  2.1  2.5  0.6]
Lasso     test R2 = 0.724    zero coefficients: 5 of 10
          [12.2  0.   0.   3.2  0.   0.   1.5  0.7  1.9  0. ]

```

We set aside 30% of the data as **test data** that the model never sees while learning. Test R squared: OLS 0.680, Lasso 0.724, Ridge 0.754. The OLS coefficients are all over the place (12.9, 0.5, -4.8, ...), Ridge is calmer, and Lasso zeroes 5 of 10. The predictors are put on one scale first (**StandardScaler**) because the penalty depends on scale. The size of the penalty is chosen automatically by cross-validation (**RidgeCV**, **LassoCV**).

⚠ WARNING

The score on **training data** is always too good. A model's value is honest only when tested on data not used to build it. This principle recurs throughout Series A3.

Non-linear regression: growth curves

Fish do not grow in a straight line forever; they approach a maximum length. The **von Bertalanffy** curve describes this: $L(t) = L_{\max} \times (1 - e^{-k(t - t_0)})$. Its parameters are not linear coefficients, so we use `curve_fit`, which searches for the best values step by step, starting from an initial guess `p0`.

```

# Chapter 27 - Non-linear regression: fish growth curve (von Bertalanffy)
import numpy as np
from scipy.optimize import curve_fit

def length(age, l_max, k, t0):
    """Fish length (cm) at a given age (months)."""
    return l_max * (1 - np.exp(-k * (age - t0)))

rng = np.random.default_rng(42)
age = np.linspace(1, 24, 30)
true = length(age, l_max=45, k=0.15, t0=-0.5)
measured = true + rng.normal(0, 1.2, age.size)    # measured length with error

estimate, covariance = curve_fit(length, age, measured, p0=[40, 0.1, 0])
error = np.sqrt(np.diag(covariance))
for name, value, se in zip(["l_max", "k", "t0"], estimate, error):
    print(f"{name:7} = {value:7.3f} (standard error {se:.3f})")

residual = measured - length(age, *estimate)
print("Root mean squared residual:", round(float(np.sqrt((residual ** 2).mean())),
3), "cm")
print("Predicted length at 12 months:", round(float(length(12, *estimate)), 1), "cm")
print("Predicted length at 36 months:", round(float(length(36, *estimate)), 1), "cm
(close to l_max)")

```

```

l_max = 45.405 (standard error 0.521)
k      = 0.146 (standard error 0.007)
t0     = -0.484 (standard error 0.165)
Root mean squared residual: 0.899 cm
Predicted length at 12 months: 38.1 cm
Predicted length at 36 months: 45.2 cm (close to l_max)

```

Estimates: $L_{\max} = 45.405$ cm (standard error 0.521), $k = 0.146$ per month (0.007), $t_0 = -0.484$. The values we planted: 45, 0.15, and -0.5 , so all three are found again within the range of the standard errors. The prediction at 36 months (45.2 cm) approaches $L_{\max}$: the curve flattens. The standard errors come from the covariance matrix returned by `curve_fit`.

⚠ WARNING

Non-linear curves can be very sensitive to the initial guess `p0` and can fail to converge. Draw the curve over the data before trusting it. Do not predict far outside the range of the data (extrapolation): the model only knows what it has seen.

Mixed-effects models: grouped data

Cattle in the same pen are not independent: they share feed, cleanliness, and a keeper. Treating them all as independent (plain OLS) deceives the standard errors. A **mixed-effects model** adds a **random effect** per group (pen) next to the **fixed effect** we want to know (feed).

```

# Chapter 27 - Mixed-effects model: cattle in many pens
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

rng = np.random.default_rng(42)
rows = []
for pen in range(1, 9):
    pen_effect = rng.normal(0, 12)          # each pen has its own fortune
    for _ in range(10):
        feed = rng.uniform(8, 14)          # kg of feed per day
        gain = 20 + 6 * feed + pen_effect + rng.normal(0, 8)
        rows.append({"pen": f"K{pen}", "feed_kg": feed, "weight_gain_kg": gain})
cattle = pd.DataFrame(rows)

plain = smf.ols("weight_gain_kg ~ feed_kg", data=cattle).fit()
mixed = smf.mixedlm("weight_gain_kg ~ feed_kg", data=cattle,
groups=cattle["pen"]).fit()

print("Plain OLS      : feed coef =", round(plain.params["feed_kg"], 3), " standard
error =", round(plain.bse["feed_kg"], 3))
print("Mixed effects : feed coef =", round(mixed.params["feed_kg"], 3), " standard
error =", round(mixed.bse["feed_kg"], 3))

pen_variance = float(mixed.cov_re.iloc[0, 0])
resid_variance = float(mixed.scale)
print("Between-pen variance:", round(pen_variance, 1))
print("Residual variance   :", round(resid_variance, 1))
print("ICC (share of variation due to the pen):", round(pen_variance / (pen_variance
+ resid_variance), 3))
print()
print("Random effect per pen (difference from the average):")
print(pd.Series({k: round(float(v.iloc[0]), 1) for k, v in
mixed.random_effects.items()})).to_string())

```

```

Plain OLS      : feed coef = 5.131  standard error = 0.772
Mixed effects : feed coef = 5.635  standard error = 0.476
Between-pen variance: 103.0
Residual variance   : 47.9
ICC (share of variation due to the pen): 0.683

```

Random effect per pen (difference from the average):

```

K1    6.5
K2   -5.7
K3   -8.0
K4    7.5
K5  -14.9
K6   -2.6
K7   16.3
K8    1.0

```

- Plain OLS estimates the effect of feed at 5.131 (standard error 0.772); the mixed model 5.635 (0.476). The value we planted was 6. The mixed model separates “the pen’s fortune” from the noise, so it is more precise.
- **Between-pen variance** 103.0 and residual variance 47.9. The **ICC** (*intraclass correlation*) = $103.0 / (103.0 + 47.9) = 0.683$: about 68% of the variation in weight gain comes from differences between pens. That is large: ignoring the pen is a serious mistake.
- The random effect per pen shows K7 (+16.3) as the luckiest and K5 (-14.9) as the unluckiest, relative to the average.

Mixed models are the standard tool for nested plots in a forest, ponds within a farm, or repeated measurements on the same tree.

Time series and forecasting with ARIMA

Data measured in sequence over time (monthly rainfall) has a memory: this month resembles last month, and there is a seasonal pattern. **ARIMA** models that through three parts: AR (past values explain the present value), I (differencing to flatten a trend), MA (past errors also have an influence). `seasonal_order=(P, D, Q, 12)` adds a pattern that repeats every 12 months.

```

# Chapter 27 - Time series and forecasting with ARIMA: monthly rainfall
import numpy as np
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA

rng = np.random.default_rng(42)
months = pd.period_range("2019-01", periods=72, freq="M")
season = 120 + 70 * np.cos(2 * np.pi * (np.arange(72) - 1) / 12)      # yearly pattern
rain = pd.Series(season + rng.normal(0, 18, 72), index=months).clip(lower=0).round(1)
# mm (fictional)

train, test = rain[:60], rain[60:]      # keep the last 12 months to test the forecast

model = ARIMA(train, order=(1, 0, 0), seasonal_order=(1, 1, 0, 12)).fit()
forecast = model.forecast(steps=12)

mae = (test - forecast).abs().mean()
# Simple benchmark: this month equals the same month last year
naive = pd.Series(train.values[-12:], index=test.index)
mae_naive = (test - naive).abs().mean()
print("MAE of the seasonal ARIMA forecast:", round(mae, 1), "mm")
print("MAE of the seasonal naive benchmark:", round(mae_naive, 1), "mm")
print()
print(pd.DataFrame({"actual": test, "forecast": forecast.round(1)}).head(6))

```

MAE of the seasonal ARIMA forecast: 15.0 mm
MAE of the seasonal naive benchmark: 17.8 mm

	actual	forecast
2024-01	150.3	187.5
2024-02	184.0	185.1
2024-03	183.6	178.4
2024-04	165.6	166.5
2024-05	132.8	108.5
2024-06	99.3	85.0

- We **keep the last 12 months** as an exam and forecast them from the 60 months before. That is the honest way to test a forecast.
- The simple benchmark (“this month = the same month last year”) misses by 17.8 mm on average; seasonal ARIMA by 15.0 mm. A complicated model must always be compared with a simple benchmark; if it does not win by a clear margin, use the simple one.
- The table shows the first six months: the forecast follows the season, but does not capture the random noise of each month (January: 187.5 forecast vs 150.3 actual). A forecast gives a pattern, not certainty.

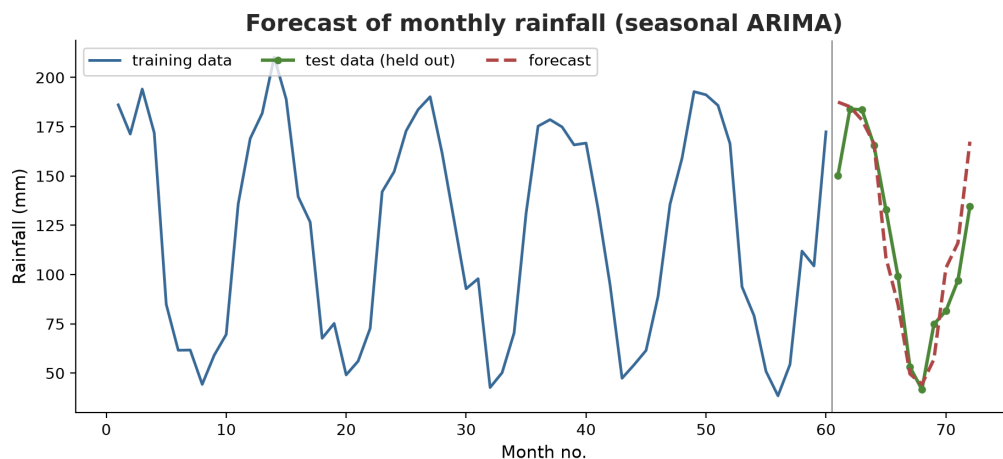


Figure 0.2: Forecasting monthly rainfall: training data, test data, and the seasonal ARIMA forecast.

Spatial regression: GWR

Ordinary regression gives **one** coefficient for the whole region. Yet the effect of rain on harvest can differ on the western and eastern slopes. **GWR** (*geographically weighted regression*) runs many local regressions, each location with its own coefficient, weighting near neighbours more heavily. The `mgwr` library does the job.

```
# Chapter 27 - Spatial regression: GWR (geographically weighted regression) with mgwr
import numpy as np
import statsmodels.api as sm
from mgwr.gwr import GWR
from mgwr.sel_bw import Sel_BW

rng = np.random.default_rng(42)
n = 150
# Coordinates of 150 orchard plots in a 10 x 10 km area (fictional)
coords = rng.uniform(0, 10, size=(n, 2))
rain = rng.normal(0, 1, size=(n, 1)) # standardised rainfall
# The effect of rain strengthens from west (x = 0) to east (x = 10)
effect = 0.5 + 0.35 * coords[:, 0]
yield_t = 5 + effect * rain + rng.normal(0, 0.4, size=(n, 1)) # tonnes/ha (fictional)
```

```

# Global model: a single coefficient for the whole area
global_ols = sm.OLS(yield_t, sm.add_constant(rain)).fit()
print("Global OLS: rain coefficient =", round(float(global_ols.params[1]), 3), " R2 =", round(float(global_ols.rsquared), 3))

# GWR: a different coefficient at every location
bandwidth = Sel_BW(coords, yield_t, rain).search()
result = GWR(coords, yield_t, rain, bandwidth).fit()
local = result.params[:, 1]
print("Chosen bandwidth (number of neighbours):", int(bandwidth))
print("GWR: R2 =", round(float(result.R2), 3))
print("Local rain coefficient: min =", round(float(local.min()), 2), " max =", round(float(local.max()), 2))

west = coords[:, 0] < 3
east = coords[:, 0] > 7
print("Mean coefficient in the west (x < 3):", round(float(local[west].mean()), 2))
print("Mean coefficient in the east (x > 7):", round(float(local[east].mean()), 2))
print("True effect in the west/east      :", round(float(effect[west].mean()), 2),
"/", round(float(effect[east].mean()), 2))

```

```

Global OLS: rain coefficient = 2.176  R2 = 0.788
Chosen bandwidth (number of neighbours): 46
GWR: R2 = 0.97
Local rain coefficient: min = 0.78  max = 3.44
Mean coefficient in the west (x < 3): 1.09
Mean coefficient in the east (x > 7): 3.26
True effect in the west/east      : 1.01 / 3.41

```

- This script uses `mgwr`, an extra library outside the basic packages of this chapter; install it with `pip install mgwr` (it brings `libpysal`, `spreg`, and `spglm` along). On the author's computer, the library was available as a ready-made package for Python 3.14 and this script was really run.
- Global OLS estimates a single rain coefficient of 2.176 (R squared 0.788). GWR finds local coefficients from 0.78 to 3.44: weak in the west (average 1.09) and strong in the east (3.26), close to the values we planted (1.01 and 3.41). One global number hides this pattern.
- `Sel_BW` chooses the **bandwidth** (here 46 nearest neighbours) through an automatic search. A narrow window = very local but noisy; a wide one = smooth but less sensitive.
- Careful: the GWR R squared of 0.97 is computed on the same data used for estimation, and GWR is far more flexible than OLS, so that figure must not simply be compared with 0.788. Use criteria such as AICc or test data. The coordinates here are fictional; real map data and projections are covered in the geospatial series.

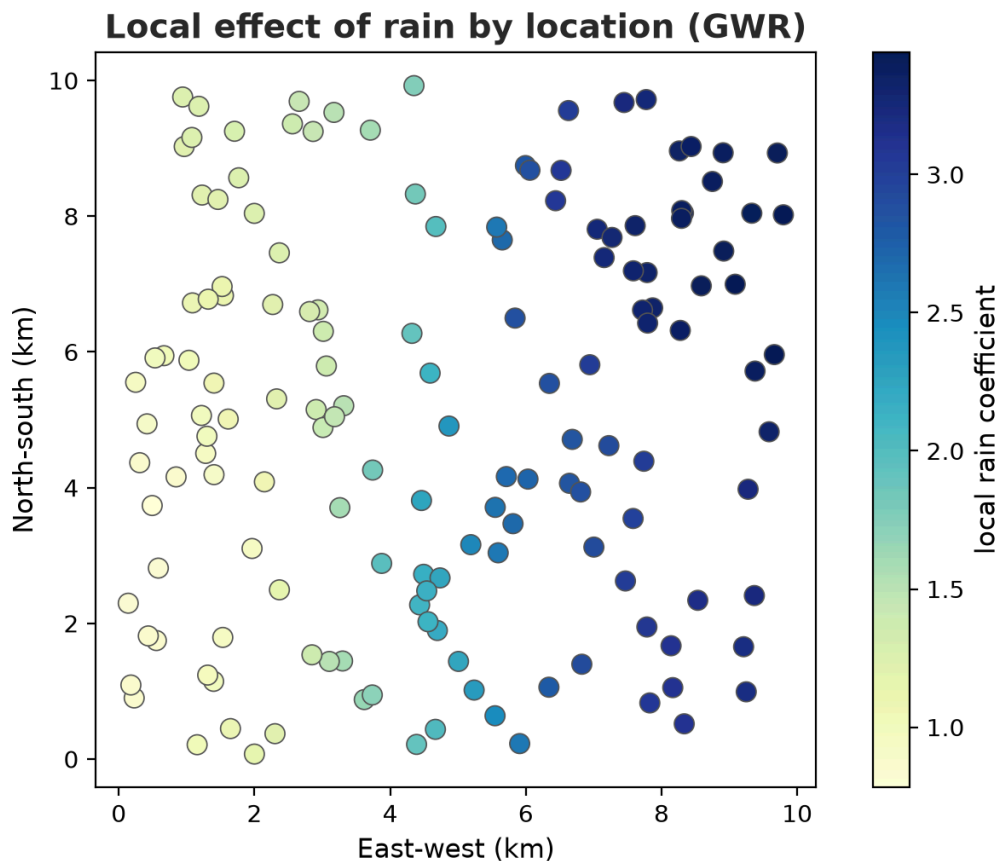


Figure 0.3: Local rain coefficients from GWR on 150 fictional orchard plots: stronger toward the east.

Monte Carlo simulation: computing uncertainty

Yield, price, and cost are uncertain. Instead of a single profit figure, we **draw** the input values hundreds of thousands of times from plausible distributions, compute the profit each time, and look at the distribution of profit. This is **Monte Carlo** (named after the casino, because of the “drawing”).

```
# Chapter 27 - Monte Carlo simulation: uncertainty in maize harvest income
import numpy as np

rng = np.random.default_rng(42)
N = 100_000
area_ha = 2.0

# Every input is uncertain, so we draw from plausible distributions (fictional example
# numbers)
yield_t_ha = rng.triangular(left=3.0, mode=5.0, right=6.5, size=N) # harvest yield
price_rp_kg = rng.normal(loc=5000, scale=700, size=N) # selling price
cost_rp = rng.normal(loc=40_000_000, scale=3_000_000, size=N) # total cost

income = yield_t_ha * 1000 * area_ha * price_rp_kg
profit = income - cost_rp

print("Mean profit (million Rp):", round(profit.mean() / 1e6, 2))
print("Standard deviation (million Rp):", round(profit.std() / 1e6, 2))
print("90% interval of profit (million Rp):", np.percentile(profit, [5,
95]).round(-4) / 1e6)
print("Probability of a loss      :", round(float((profit < 0).mean()), 3))
```

```
# How many draws are enough? Compare a few sizes
for n in (100, 1000, 100_000):
    print(f" loss probability estimated from {n:>7} draws:", round(float((profit[:n]
< 0).mean()), 3))
```

```
Mean profit (million Rp): 8.36
Standard deviation (million Rp): 10.34
90% interval of profit (million Rp): [-8.03 25.96]
Probability of a loss      : 0.216
  loss probability estimated from      100 draws: 0.21
  loss probability estimated from     1000 draws: 0.224
  loss probability estimated from 100000 draws: 0.216
```

The mean profit is 8.36 million rupiah, but its standard deviation is 10.34 million: the uncertainty is large. The 90% interval of profit: from -8.03 to 25.96 million. The **probability of a loss** is 21.6%. A single average hides this risk. The last three lines show the estimated loss probability settling around 0.2 after thousands of draws; 100 draws is too few for a tidy number. All numbers (yield, price, cost) are fictional examples, not real prices or advice.

Optimisation: finding the best choice

`scipy.optimize` solves two common kinds of problem. **Linear programming** (`linprog`): minimise cost subject to linear equalities and inequalities. **One-variable optimisation** (`minimize_scalar`): find the highest or lowest point of a function.

```
# Chapter 27 - Optimisation with scipy.optimize: the cheapest feed mix
from scipy.optimize import linprog, minimize_scalar

# Two ingredients: maize (Rp 5,000/kg, 8% protein) and soybean meal (Rp 9,000/kg,
44% protein)
# Requirement: 100 kg of mix with at least 18% protein. Fictional example numbers.
cost = [5000, 9000]
# linprog wants the "less than" form: -(protein) <= -(requirement)
protein_a = [[-0.08, -0.44]]
protein_b = [-0.18 * 100]
total_a = [[1, 1]]
total_b = [100]

result = linprog(cost, A_ub=protein_a, b_ub=protein_b,
                  A_eq=total_a, b_eq=total_b, bounds=[(0, None), (0, None)])
print("Success?          :", result.success)
print("Maize (kg)        :", round(result.x[0], 2))
print("Soybean (kg)       :", round(result.x[1], 2))
print("Cost (Rp)          :", round(result.fun))

# One-variable optimisation: the fish stocking density that maximises profit (fictional
model)
def negative_profit(density):
    final_weight = 900 * (1 - density / 120)      # weight shrinks when too crowded
    return -(density * final_weight / 1000 * 25000 - density * 800)

search = minimize_scalar(negative_profit, bounds=(1, 100), method="bounded")
print()
print("Best stocking density (fish/m2):", round(search.x, 1))
print("Maximum profit (Rp/m2)         :", round(-search.fun))
```

```
Success?          : True
Maize (kg)        : 72.22
```

```
Soybean (kg)      : 27.78
Cost (Rp)         : 611111

Best stocking density (fish/m2): 57.9
Maximum profit (Rp/m2)      : 627853
```

- A 100 kg feed mix with at least 18% protein uses 72.22 kg of maize and 27.78 kg of soybean meal, at a cost of Rp 611,111. Soybean meal is pricier but rich in protein; the solver finds the cheapest mix that meets the requirement. `linprog` only accepts the “less than or equal” form, so the “minimum protein” requirement is written with a minus sign.
- The second problem: the stocking density that maximises profit is 57.9 fish/m2. Too sparse wastes the pond; too crowded stunts the fish. The optimum lies in the middle. The model is deliberately simple and fictional.

System dynamics models and sensitivity analysis

Systems that change over time (a fish stock, a forest stand) are often written as equations of **rate of change**. The **logistic** model says the stock grows by $r \times \text{stock} \times (1 - \text{stock} / K)$ (K = carrying capacity), minus the harvest. `solve_ivp` computes the journey step by step.

```
# Chapter 27 - System dynamics and sensitivity analysis: fish stock in a small reservoir
import numpy as np
from scipy.integrate import solve_ivp

def final_stock(r=0.8, K=1000.0, harvest=100.0, start=300.0, years=10):
    """Logistic population with a fixed harvest per year (tonnes, fictional numbers)."""
    def rate(t, y):
        return [r * y[0] * (1 - y[0] / K) - harvest]
    extinct = lambda t, y: y[0] # event: the stock touches zero
    extinct.terminal = True
    sol = solve_ivp(rate, (0, years), [start], events=extinct, t_eval=[years])
    return 0.0 if sol.status == 1 else float(sol.y[0, -1])

baseline = final_stock()
print("Stock after 10 years (baseline case):", round(baseline, 1), "tonnes")

# One-at-a-time sensitivity analysis: change each parameter by +-20%
print()
print(f"{'parameter':10} {'-20%':>8} {'+20%':>8} {'range':>9}")
for name in ("r", "K", "harvest", "start"):
    values = dict(r=0.8, K=1000.0, harvest=100.0, start=300.0)
    low = final_stock(**values, name: values[name] * 0.8)
    high = final_stock(**values, name: values[name] * 1.2)
    print(f"{name:10} {low:8.1f} {high:8.1f} {abs(high - low):9.1f}")

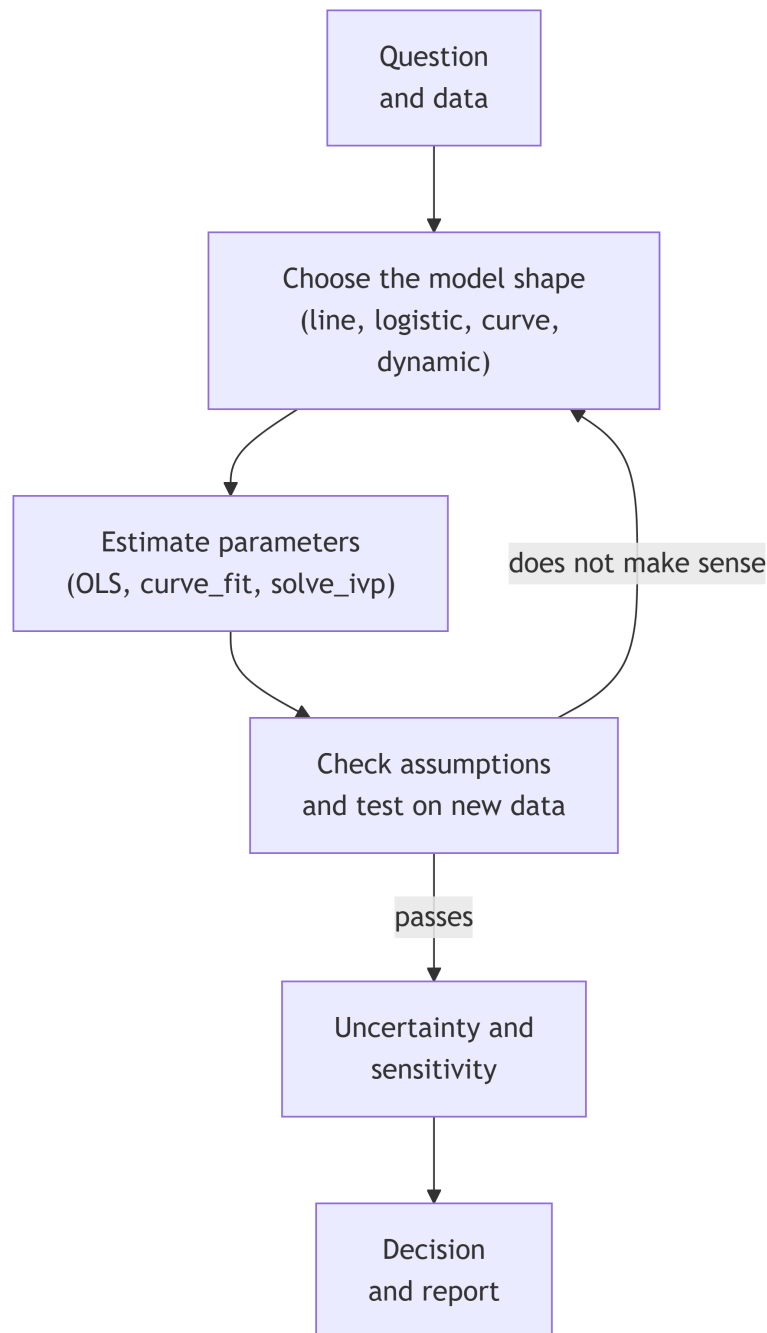
# Tipping point: which harvest is still sustainable?
for harvest in (100, 150, 190, 210):
    print(f"harvest    {harvest}      tonnes/year    ->    final    stock
{final_stock(harvest=harvest):7.1f} tonnes")
```

Stock after 10 years (baseline case): 844.8 tonnes

parameter	-20%	+20%	range
r	753.3	880.2	126.9
K	636.5	1048.6	412.1
harvest	882.4	798.9	83.5
start	837.7	847.9	10.2

harvest 100 tonnes/year	-> final stock	844.8 tonnes
harvest 150 tonnes/year	-> final stock	679.6 tonnes
harvest 190 tonnes/year	-> final stock	0.0 tonnes
harvest 210 tonnes/year	-> final stock	0.0 tonnes

- Baseline case: a stock of 300 tonnes rises to 844.8 tonnes after 10 years with a harvest of 100 tonnes per year.
- A **sensitivity analysis** changes one parameter by $\pm 20\%$ while holding the others, then sees how far the result shifts. The carrying capacity K matters most (range 412.1 tonnes), the rate r moderately (126.9), the harvest little (83.5), and the starting stock hardly at all (10.2) because the stock heads for equilibrium. This result shows which data deserves careful measurement: **K**.
- **Tipping point**: a harvest of 150 tonnes/year is still safe (stock 679.6), but 190 tonnes/year **destroys the stock** (0.0), even though that figure is below the theoretical limit $rK/4 = 200$ tonnes. The reason: with a harvest of 190 the stock balance equation has two points, about 388 tonnes (unstable) and 612 tonnes (stable). The starting stock of 300 lies below the unstable point, so the stock slides down until it is gone. Real fish stocks can collapse suddenly. That is why system models are useful.



The quantitative modelling loop: from question to decision, circling back when results do not make sense.

Running Project I3: models for our information system

We bring together the project's four models: tree allometry, fish weight against feed, cattle weight against feed, and harvest against rainfall. The `summarise()` function condenses each model into one row.

```
# Chapter 27 - Running project I3: four models for our information system
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
```

```

rng = np.random.default_rng(42)

def summarise(name, model, variable):
    """One summary row: coefficient, 95% interval, R squared."""
    b = model.params[variable]
    lo, hi = model.conf_int().loc[variable]
    return {"model": name, "variable": variable, "coef": round(b, 3),
            "lower95": round(lo, 3), "upper95": round(hi, 3), "R2": round(model.rsquared,
3)}

# 1. Tree allometry: ln(biomass) ~ ln(diameter)
d = rng.uniform(10, 60, 60)
trees = pd.DataFrame({"ln_d": np.log(d), "ln_b": np.log(0.12 * d ** 2.4 *
np.exp(rng.normal(0, 0.15, 60))))})
m1 = smf.ols("ln_b ~ ln_d", data=trees).fit()

# 2. Fish: weight gain (g) ~ feed (g/day)
fish_feed = rng.uniform(20, 60, 50)
fish = pd.DataFrame({"feed": fish_feed, "gain_g": 15 + 4.2 * fish_feed + rng.normal(0,
20, 50)})
m2 = smf.ols("gain_g ~ feed", data=fish).fit()

# 3. Cattle: weight gain (kg) ~ feed (kg/day)
cattle_feed = rng.uniform(8, 14, 40)
cattle = pd.DataFrame({"feed": cattle_feed, "gain_kg": 10 + 5.5 * cattle_feed +
rng.normal(0, 7, 40)})
m3 = smf.ols("gain_kg ~ feed", data=cattle).fit()

# 4. Harvest vs rainfall: a humped curve (too dry or too wet both hurt)
rain = rng.uniform(600, 2400, 70) # mm per season
harvest = 0.5 + 0.0060 * rain - 0.0000018 * rain ** 2 + rng.normal(0, 0.3, 70) #
tonnes/ha
orchard = pd.DataFrame({"rain": rain, "harvest": harvest})
m4 = smf.ols("harvest ~ rain + I(rain ** 2)", data=orchard).fit()

table = pd.DataFrame([
    summarise("Tree allometry (log scale)", m1, "ln_d"),
    summarise("Fish: weight ~ feed", m2, "feed"),
    summarise("Cattle: weight ~ feed", m3, "feed"),
])
print(table.to_string(index=False))

b1, b2 = m4.params["rain"], m4.params["I(rain ** 2)"]
optimum = -b1 / (2 * b2)
print()
print("Harvest ~ rain + rain^2: R2 =", round(m4.rsquared, 3), " squared-term p =",
format(m4.pvalues["I(rain ** 2)"], ".1e"))
print("Optimal rainfall (peak of the curve):", round(optimum), "mm per season")

# Uncertainty of the peak via bootstrap (resampling with replacement)
peaks = []
for _ in range(500):
    s = orchard.sample(len(orchard), replace=True,
random_state=int(rng.integers(1_000_000)))
    mb = smf.ols("harvest ~ rain + I(rain ** 2)", data=s).fit()
    peaks.append(-mb.params["rain"] / (2 * mb.params["I(rain ** 2)"]))

```

```
print("95% bootstrap interval for the peak:", np.percentile(peaks, [2.5,
97.5])).round().tolist(), "mm")
```

	model	variable	coef	lower95	upper95	R2
Tree allometry (log scale)		ln_d	2.397	2.333	2.461	0.990
Fish: weight ~ feed		feed	4.210	3.753	4.667	0.877
Cattle: weight ~ feed		feed	5.917	4.788	7.045	0.748

```
Harvest ~ rain + rain^2: R2 = 0.608 squared-term p = 1.6e-10
Optimal rainfall (peak of the curve): 1711 mm per season
95% bootstrap interval for the peak: [1653.0, 1800.0] mm
```

- The table: allometry $b = 2.397$ (R squared 0.990), fish 4.210 g per gram of feed (0.877), cattle 5.917 kg per kg of feed (0.748). The confidence intervals contain the values we planted (2.4; 4.2; 5.5). The cattle interval is the widest (4.788 to 7.045) because it has the least data (40) and is the noisiest.
- Harvest versus rain has a **humped curve**: too dry or too wet both hurt. The model `harvest ~ rain + I(rain ** 2)` is still **linear in its coefficients**, so plain OLS is enough. The squared term is highly significant ($p = 1.6e-10$), R squared 0.608.
- The peak of the curve, $-b_1 / (2 \times b_2)$, is at about **1711 mm** per season (the value we planted: 1667). The bootstrap (500 resamples with replacement) gives a 95% interval for the peak: 1653 to 1800 mm. Giving an “optimum” number an uncertainty interval is far more honest than a single figure.

All these models go into the Quarto report in Chapter 29.

Exercises

1. In `01_ols_allometric.py`, predict the biomass of a 25 cm and a 55 cm tree (with bias correction). Why is predicting a 150 cm tree not safe?
2. In `03_logistic.py`, change the diameter coefficient in the data from 0.12 to 0.02. How do the accuracy and the odds ratio change?
3. In `12_system_dynamics.py`, find (by trying several values) the largest harvest for which the stock still survives after 10 years. Compare it with 200 tonnes.
4. **Challenge**: create data with one important predictor and five junk predictors, then compare the test R squared of OLS and Lasso. When does Lasso win?

Chapter Summary

- **OLS**: `smf.ols("y ~ x", data=...)`; read the coefficients, the 95% interval, R squared. Log-log turns a power relationship into a straight line.
- **Diagnostics**: normal residuals (Shapiro), equal variance (Breusch-Pagan), no twins (VIF).
- **Logistic** for yes/no (odds ratios), **Poisson GLM** for counts (check overdispersion), **Ridge/Lasso** for many predictors (test on test data).
- **Non-linear** (`curve_fit`), **mixed effects** (`mixedlm`) for grouped data, **ARIMA** for time series (test with held-out data), **GWR** for coefficients that change by location.
- **Monte Carlo** computes uncertainty; **optimisation** finds the best choice; **system dynamics** and **sensitivity** show tipping points and the most influential variables.
- A model is a tool, not the truth: check assumptions, test on new data, and report the uncertainty.

Next, in Chapter 28: qualitative and mixed methods: how to analyse interviews with farmers, fishers, and livestock keepers, and combine them with numbers.

Chapter 28 — Qualitative and Mixed Methods: Hearing the Stories, Then Counting Them

Series I3 — Statistics and Modelling

In this chapter, you will:

- Understand when to use qualitative, quantitative, or mixed methods
- Code interviews into themes with a codebook, then count how often they appear
- Use basic text tools: word frequency, KWIC, TF-IDF, topic modelling, and dictionary-based sentiment
- Understand embeddings as a concept and turn qualitative data into numbers
- Judge the validity, reliability, and ethics of research, then combine stories and numbers in mixed methods

All scripts for this chapter are in the `code/ch28/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Chapters 26 and 27 dealt with numbers. But if you want to know **why** farmers are reluctant to plant trees on their land, or **how** fishers cope with the lean season, numbers alone are not enough. You need to **hear the stories**. Imagine a veterinarian: a cow's temperature says the cow has a fever, but the keeper's story ("yesterday my cow refused to eat, and my neighbour lost two animals") says why and what to look for. **Qualitative** methods analyse words, meaning, and stories; **mixed methods** combine both.

⚠ WARNING

Ethics note and made-up data. All respondents, interview quotes, and survey numbers in this chapter are **fictional**, written by the book author for practice, and do not represent any person or village. In real research you must: (1) obtain the respondents' **consent** after explaining the purpose and risks (*informed consent*, ideally written); (2) **disguise identities** (anonymisation) in data and reports; (3) store recordings and transcripts securely; (4) explain that respondents may stop at any time. For indigenous and local communities, there is also a principle of community consent sought beforehand and without pressure, which you should study from your ethics committee's guidelines. Also check your institution's or university's rules on research permits.

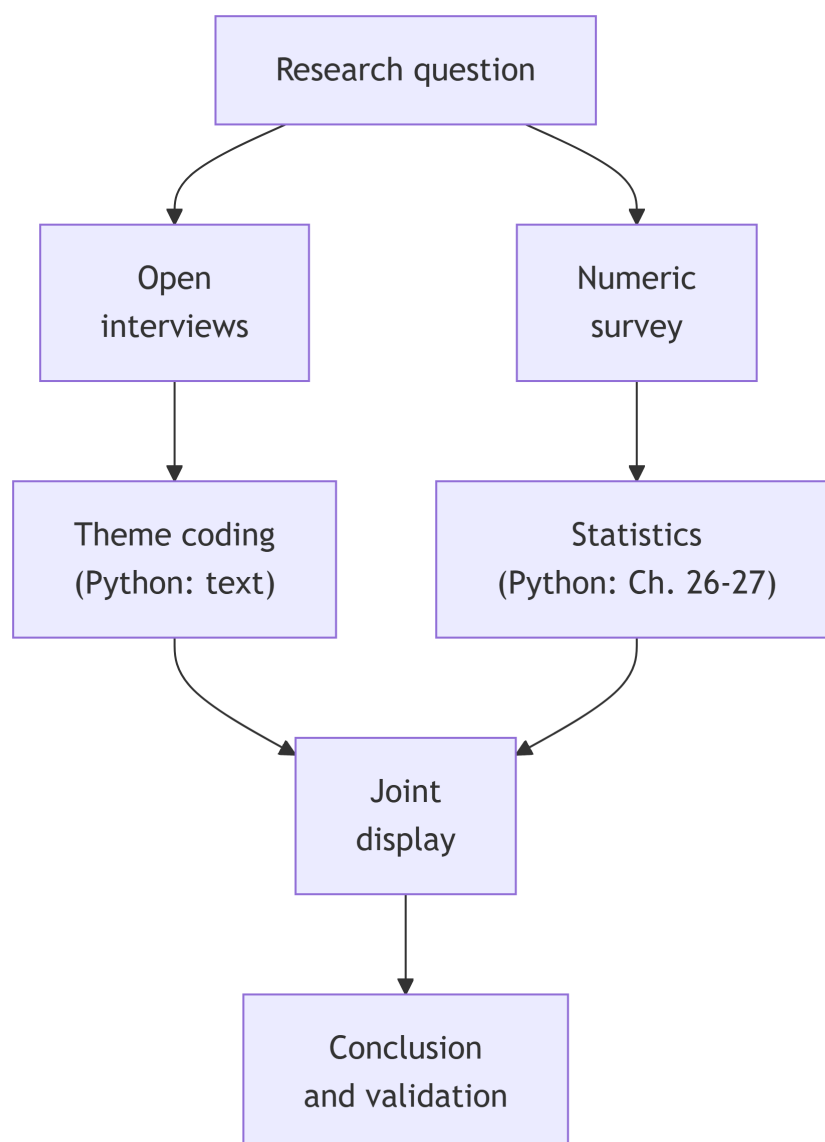
Qualitative, quantitative, or mixed?

	Quantitative	Qualitative
Typical question	How many? How big? Is there a difference?	Why? How? What does it mean?
Data	Numbers (weight, harvest, score 1-5)	Text, stories, observations, documents
Sample	Large, random	Small, chosen on purpose
Main tools	Hypothesis tests, regression (Chapters 26-27)	Theme coding, content analysis

	Quantitative	Qualitative
Strength	Can be generalised, measurable	Rich in context, finds the unexpected
Weakness	Shallow on “why”	Hard to generalise, prone to subjectivity

A rule of thumb: **quantitative for “how many”**, **qualitative for “why”**, and both when the decision is important. There are three common **mixed designs**:

- **Explanatory sequential**: a numeric survey first, then interviews to explain surprising findings.
- **Exploratory sequential**: interviews first to discover the topics, then a survey to measure how common they are.
- **Convergent** (parallel): both are collected at the same time and then compared. Our project uses this.



A convergent mixed design: two data tracks collected together, then merged into one conclusion.

Our made-up interview data

We use 24 short fictional interviews: 8 farmers, 8 fishers, 8 livestock keepers. The open question: “*What is the biggest change you have felt in the last five years, and how did you deal with it?*” Each respondent also filled in a **satisfaction** score (1 to 5) and an **income** (million rupiah per month), so we have both a story and numbers for the same person. The data is in `kode/ch28/data/interviews.csv`.

```
# Chapter 28 - Reading made-up (fictional) interview data
from pathlib import Path

import pandas as pd

# ETHICS NOTE: all respondents and quotes in this chapter are FICTIONAL, written by
# the book author.
# In real research you must obtain written informed consent, disguise identities
# (anonymisation),
# and store recordings securely.
data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

print("Number of respondents:", len(data))
print(data["role"].value_counts().to_string())
print()
print("Average satisfaction (scale 1-5) by role:")
print(data.groupby("role")["satisfaction"].mean().round(2).to_string())
print()
data["word_count"] = data["answer"].str.split().str.len()
print("Answer length (words): min", data["word_count"].min(), " max",
      data["word_count"].max(),
      " mean", round(data["word_count"].mean(), 1))
print()
for _, row in data.head(3).iterrows():
    print(f"[{row['code']}] {row['role']} {row['age']} y] {row['answer']}")
```

Number of respondents: 24

role	
farmer	8
fisher	8
livestock keeper	8

Average satisfaction (scale 1-5) by role:

role	
farmer	2.75
fisher	2.50
livestock keeper	3.00

Answer length (words): min 21 max 32 mean 26.0

[R01, farmer, 45 y] The rain is unpredictable now. Last year my rice field ran dry right after planting, and the harvest dropped by almost half. I worry it will happen again this year.

[R02, farmer, 52 y] Fertiliser keeps getting more expensive, yet the price of rice drops at harvest time. Middlemen come with cheap offers and we have to sell because we need cash.

[R03, farmer, 38 y] Our farmer group helps. We buy fertiliser together so it is cheaper, and the extension officer visits every month. This year's harvest is good and I am happy.

Each group has 8 people. Average satisfaction: fishers 2.50, farmers 2.75, livestock keepers 3.00. The answers are short, 21 to 32 words (real interviews are far longer). The first three quotes show the variety: drought, a low price from a middleman, and a farmer group that helps.

Theme coding

Coding means attaching a label to pieces of text that talk about the same thing. The label is called a **code** or **theme**. The list of themes with their definitions is called a **codebook**. There are two ways to build it:

- **Inductive:** themes emerge from the data; you read first, then name the repeating patterns.
- **Deductive:** themes come from theory or the research question, then are looked for in the data.

In practice researchers mix them: start inductively from part of the transcripts, build a codebook, then apply it deductively to all the data. Python helps with that last step. The simplest way: each theme has a list of **keywords**.

```
# Chapter 28 - Theme coding with a simple codebook
import re
from pathlib import Path

import pandas as pd

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

# Codebook: theme -> keywords (word beginnings). Built by the researcher after reading
# part of the data.
codebook = {
    "climate": ["rain", "dry", "weather", "flood", "waves", "lean season"],
    "price_market": ["price", "middlem", "collector", "sell", "margin"],
    "cost_capital": ["capital", "loan", "interest", "expensive", "cost", "diesel",
"fertili", "feed"],
    "pest_disease": ["pest", "planthopper", "disease", "vaccine", "died", "bombing"],
    "cooperation": ["group", "cooperat", "working together", "together", "share"],
    "technology_info": ["app", "phone", "video", "information", "training", "extension
officer", "guidance"],
    "rules_land": ["permit", "forest area", "evicted", "rules", "zone", "certificate",
"officers"],
}

def assign_codes(text):
    """List of themes whose keyword appears in the text (lower case, word beginning)."""
    text = text.lower()
    return [theme for theme, keys in codebook.items()
            if any(re.search(r"\b" + re.escape(k), text) for k in keys)]

data["themes"] = data["answer"].apply(assign_codes)
for theme in codebook:
    data[theme] = data["themes"].apply(lambda items: theme in items)

print("Number of respondents mentioning each theme (out of", len(data),
"respondents):")
print(data[list(codebook)].sum().sort_values(ascending=False).to_string())
print()
print("Average themes per respondent:", round(data["themes"].str.len().mean(), 2))
print("Respondents with no theme at all:", int((data["themes"].str.len() == 0).sum()))
print()
for code in ("R01", "R03", "R09"):
    row = data[data["code"] == code].iloc[0]
    print(code, "->", row["themes"])
```

```
Number of respondents mentioning each theme (out of 24 respondents):
cost_capital      11
price_market      9
climate           7
```

```

technology_info    6
cooperation        5
pest_disease       4
rules_land         3

```

Average themes per respondent: 1.88

Respondents with no theme at all: 0

```
R01 -> ['climate']
```

```
R03 -> ['cost_capital', 'cooperation', 'technology_info']
```

```
R09 -> ['climate']
```

- `re.search(r"\b" + word, text)` looks for a keyword **starting** at a word boundary. `\b` means “word boundary”, learned in Chapter 22.
- The most common theme is `cost_capital` (11 of 24 respondents), then `price_market` (9) and `climate` (7). The rarest theme: `rules_land` (3).
- On average 1.88 themes per respondent: one answer often talks about several things, so the number of themes is larger than the number of respondents. R03 gets three themes: `cost_capital`, `cooperation`, and `technology_info`.

⚠ WARNING

Keyword-based coding is fast and consistent but **crude**. A keyword like `rain` also catches “rainfall” and “rainy” (good), but `expensive` is always counted as cost even when it is about something else, and a person who writes “drought” without the word “dry” is missed. That is why machine results need to be **re-read by a human**. Treat automatic codes as a fast but careless assistant, not a judge.

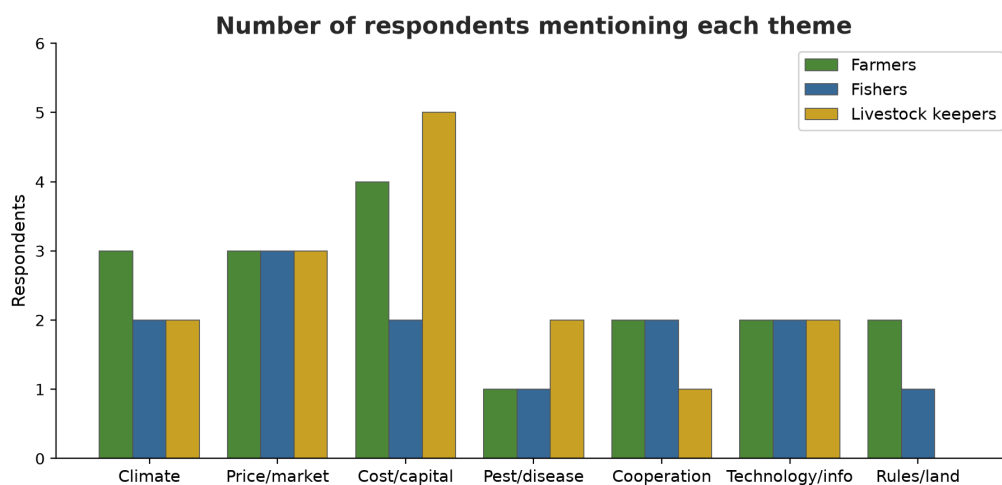


Figure 0.1: Number of respondents mentioning each theme, by role. Costs and capital are mentioned by many groups; land rules are more typical of farmers.

Content analysis: word frequency and KWIC

Content analysis systematically counts and groups elements of text. Two basic tools: **word frequency** (after removing common words such as “the”, “and”, called *stopwords*) and **KWIC** (*keyword in context*), which shows a keyword with a few words to its left and right so you can see how the word is used.

```

# Chapter 28 - Content analysis: word frequency and KWIC (keyword in context)
import re
from collections import Counter
from pathlib import Path

```

```

import pandas as pd

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

COMMON = set("""the and for are but with that this have has had our not was were from
its they their them
when which what your you also all can will would one two out into than then too very
much many more
some any each just only now last year because while since before after over under again
still even
until about there here been being does did done like get got may might must should
could every both
off her his him she him how why who whom those these other another such own
same""").split())

def words(text):
    return [w for w in re.findall(r"[a-z]+", text.lower()) if w not in COMMON and
len(w) > 2]

total = Counter()
for answer in data["answer"]:
    total.update(words(answer))

print("15 most frequent words (after removing common words):")
for word, n in total.most_common(15):
    print(f" {word:12} {n}")

# KWIC: show the keyword with 4 words to the left and right
print()
print("Context of the word 'price':")
for code, answer in zip(data["code"], data["answer"]):
    tokens = re.findall(r"[A-Za-z]+", answer)
    for i, t in enumerate(tokens):
        if t.lower() == "price":
            left = " ".join(tokens[max(0, i - 4):i])
            right = " ".join(tokens[i + 1:i + 5])
            print(f" {code}: ...{left} [{t}] {right}...")
            break

```

```

15 most frequent words (after removing common words):
fish          9
price         7
helps        6
feed         6
hard         5
together     4
good         4
high         4
costs        4
often        4
dry          3
harvest      3
dropped      3
worry        3
getting      3

```

```

Context of the word 'price':

```

```

R02: ...more expensive yet the [price] of rice drops at...
R11: ...fish at a stable [price] and provides ice Since...
R12: ...The collector sets the [price] and often squeezes us...
R13: ...are fresher and the [price] is better...
R17: ...very expensive and the [price] keeps rising My cows...
R21: ...up and the selling [price] is good That cooperation...

```

The most frequent words: fish (9), price (7), helps (6), feed (6). Frequency gives a **hint**, not an answer: “price” appears often, but KWIC shows the contexts are very different. R02: the price of rice drops (bad); R11: a stable price through the cooperative (good); R17: the price of feed keeps rising. Same word, opposite meanings. That is why content analysis must not stop at counts.

In **policy document analysis** (for example forest management regulations), the same tools are used: count keywords such as “community”, “permit”, and “sanction”, then compare with KWIC to see who is named as the subject and who as the object. **Discourse analysis** goes further: it asks how word choices shape a picture of who has authority and who is to blame. This part is more careful reading than counting, and Python only helps find the passages worth reading.

TF-IDF: distinctive words

A frequent word is not necessarily **distinctive**. “Fish” appears often among fishers, but what sets fishers apart from farmers? **TF-IDF** (*term frequency-inverse document frequency*) gives a high score to words that are **frequent in this document but rare in the others**. Here, each role group is joined into one “document”.

```

# Chapter 28 - TF-IDF: words that are distinctive for each group of respondents
from pathlib import Path

import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

COMMON = """the and for are but with that this have has had our not was were from its
they their them
when which what your you also all can will would one two out into than then too very
much many more
some any each just only now last year because while since before after over under again
still even
until about there here been being does did done like get got may might must should
could every both
off her his him she how why who whom those these other another such own same""".split()

# Join all answers per role into one "document"
per_role = data.groupby("role")["answer"].apply(" ".join)

vectoriser = TfidfVectorizer(stop_words=COMMON, token_pattern=r"[a-z]{3,}",
                             lowercase=True)
matrix = vectoriser.fit_transform(per_role)
words = vectoriser.get_feature_names_out()

print("Matrix size (roles x words):", matrix.shape)
for i, role in enumerate(per_role.index):
    scores = pd.Series(matrix[i].toarray()[0],
                        index=words).sort_values(ascending=False)
    print(f"{role:17}", " ", ".join(f'{w} ({s:.2f})' for w, s in scores.head(5).items()))

```

```

Matrix size (roles x words): (3, 230)
farmer      harvest (0.28), fertiliser (0.19), middlemen (0.19), planting (0.19),
land (0.19)

```

```
fisher          fish (0.67), price (0.18), fishing (0.15), reef (0.15), income (0.15)
livestock keeper feed (0.51), cattle (0.17), fall (0.17), livestock (0.17), thin (0.17)
```

The matrix is 3 roles x 230 words. Distinctive for fishers: fish, price, fishing, reef, income. Farmers: harvest, fertiliser, middlemen, planting, land. Livestock keepers: feed, cattle, livestock. A high TF-IDF score means “a distinguishing mark of this group”. It is a quick way to draw an initial picture before reading in depth.

Topic modelling: topics from a pile of text

When the text runs to thousands of pages, reading it all is impossible. **Topic modelling** looks for hidden **topics**: groups of words that tend to appear together. We use **NMF** (*non-negative matrix factorization*) from scikit-learn on the TF-IDF matrix. A popular alternative is **LatentDirichletAllocation** (LDA) in the same library; both are used similarly. We choose the number of topics (`n_components`) ourselves.

```
# Chapter 28 - Topic modelling with NMF (scikit-learn)
from pathlib import Path

import numpy as np
import pandas as pd
from sklearn.decomposition import NMF
from sklearn.feature_extraction.text import TfidfVectorizer

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

COMMON = """the and for are but with that this have has had our not was were from its
they their them
when which what your you also all can will would one two out into than then too very
much many more
some any each just only now last year because while since before after over under again
still even
until about there here been being does did done like get got may might must should
could every both
off her his him she how why who whom those these other another such own same""".split()

vectoriser = TfidfVectorizer(stop_words=COMMON, token_pattern=r"[a-z]{3,}", min_df=2)
X = vectoriser.fit_transform(data["answer"])
words = vectoriser.get_feature_names_out()
print("Words used (appearing in >= 2 answers):", len(words))

model = NMF(n_components=4, init="nndsvd", random_state=42, max_iter=500)
doc_weights = model.fit_transform(X) # respondents x topics

for t, component in enumerate(model.components_):
    top = np.argsort(component)[-1][6]
    print(f"Topic {t + 1}:", ", ".join(words[top]))

data["topic"] = doc_weights.argmax(axis=1) + 1
print()
print("Number of respondents dominated by each topic:")
print(data["topic"].value_counts().sort_index().to_string())
print()
print(pd.crosstab(data["topic"], data["role"]).to_string())
```

```
Words used (appearing in >= 2 answers): 53
Topic 1: fish, hard, capital, damaged, costs, season
Topic 2: afraid, lost, land, high, money, lot
Topic 3: price, often, lose, expensive, sell, catch
Topic 4: helps, good, together, group, livestock, lot
```

Number of respondents dominated by each topic:

topic	
1	3
2	6
3	6
4	9

role	farmer	fisher	livestock keeper
topic			
1	0	3	0
2	3	1	2
3	1	3	2
4	4	1	4

Only words appearing in at least 2 answers are used (53 words). The four topics found, with their top words:

- Topic 1: *fish, hard, capital, damaged, costs, season*: about **hard seasons and costs** (mostly fishers).
- Topic 2: *afraid, lost, land, high, money, lot*: about **fear and losses**.
- Topic 3: *price, often, lose, expensive, sell, catch*: about **prices and losing money**.
- Topic 4: *helps, good, together, group, livestock*: about **support and shared success**.

The topic names are **our** interpretation, not machine output. With only 24 short answers the topics are crude and will change if you alter the number of topics or the random seed. Topic modelling is suited to **exploring** large piles of text, not to proving something. The cross table shows topic 4 (support) is the one most respondents are dominated by (9 of 24).

Dictionary-based sentiment

Sentiment analysis judges whether a text is positive or negative in tone. The simplest way uses a **dictionary**: a list of positive and negative words, then score = positives minus negatives. We compare it with the satisfaction score the respondents filled in themselves, to test whether our dictionary makes sense.

```
# Chapter 28 - Sentiment analysis with a simple lexicon
from pathlib import Path
import re

import pandas as pd
from scipy import stats

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

POSITIVE = {"helps", "good", "happy", "satisfied", "improved", "up", "smoothly",
            "stable", "fresher",
            "healthy", "better", "plentiful", "cheaper", "help", "superior", "helped"}
NEGATIVE = {"worry", "lose", "lost", "hard", "difficult", "expensive", "dropped",
            "drops", "afraid", "died",
            "damaged", "disappearing", "less", "thin", "slow", "confusing", "squeezes",
            "decreased", "declining",
            "falls", "fall", "dry", "damage"}

def sentiment_score(text):
    words = re.findall(r"[a-z]+", text.lower())
    return sum(w in POSITIVE for w in words) - sum(w in NEGATIVE for w in words)

data["sentiment"] = data["answer"].apply(sentiment_score)
data["label"] = pd.cut(data["sentiment"], bins=[-99, -1, 0, 99], labels=["negative",
                                "neutral", "positive"])
```



```

print(data["label"].value_counts().reindex(["negative",
"neutral", "positive"]).to_string())
print()
print("Average satisfaction per sentiment label:")
print(data.groupby("label",
observed=True)["satisfaction"].mean().round(2).to_string())

rho, p = stats.spearmanr(data["sentiment"], data["satisfaction"])
print()
print("Correlation lexicon sentiment vs satisfaction score (Spearman):", round(rho,
3), " p =", format(p, ".1e"))

# Weakness of a lexicon: find respondents whose label contradicts their satisfaction
score
wrong = data[((data["label"] == "positive") & (data["satisfaction"] <= 3))
| ((data["label"] == "negative") & (data["satisfaction"] >= 4))]
print()
print("Label contradicts satisfaction:", len(wrong), "out of", len(data))

# In this tidy made-up data the lexicon is nearly perfect. In the real world it is
easily fooled:
for sentence in ("The harvest is not good", "Feed keeps going up", "Prices went up, I
am happy"):
    print(f" '{sentence}' -> score {sentiment_score(sentence)}")

```

```

label
negative    17
neutral      0
positive     7

```

Average satisfaction per sentiment label:

```

label
negative    2.06
positive    4.43

```

Correlation lexicon sentiment vs satisfaction score (Spearman): 0.738 p = 3.9e-05

Label contradicts satisfaction: 0 out of 24

```

'The harvest is not good' -> score 1
'Feed keeps going up' -> score 1
'Prices went up, I am happy' -> score 2

```

- 17 answers are negative in tone and 7 positive. Average satisfaction: 2.06 for the negative ones and 4.43 for the positive ones. The Spearman correlation between the dictionary score and satisfaction is 0.738 ($p = 3.9e-05$): our dictionary agrees with the respondents' own judgement. Note that this is **validation**: we test the tool against comparison data.
- In this tidy made-up data no label contradicts satisfaction. In the real world a dictionary is easily fooled. Three test sentences show the weakness: "The harvest is **not good**" scores +1 (the dictionary does not understand negation), "Feed keeps going **up**" scores +1 (up is good for income, bad for costs), and sarcasm goes undetected.

TIP

Ready-made sentiment dictionaries are limited for many languages, and word meaning depends on the field (in agriculture, "up" can be good or bad). Build a dictionary specific to your field, test it on a sample rated by humans, and report the agreement.

Embeddings: words become points on a map of meaning

A computer does not understand words, only numbers. An **embedding** turns text into a row of numbers (a **vector**) so that texts of similar meaning have vectors close together. Picture a map: cities that are geographically close are close on the map. An embedding makes a “map of meaning”: “flood” and “heavy rain” lie close together, “vaccine” is far from “diesel”.

Large language models produce high-quality embeddings, but those models are big and **not installed in this book**; they are covered in Series A4. Here we make a **mini embedding** the classic way: TF-IDF, then compressed by **TruncatedSVD** (also called *latent semantic analysis*, LSA). The principle is the same: text becomes a short vector, then similarity is measured with **cosine similarity** (1 = same direction, 0 = at right angles).

```
# Chapter 28 - The embedding concept: text becomes a vector, distance = similarity
of meaning
from pathlib import Path

import numpy as np
import pandas as pd
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

# Step 1: text -> TF-IDF vector (one number per word, mostly zeros)
tfidf = TfidfVectorizer(token_pattern=r"[a-z]{3,}")
X = tfidf.fit_transform(data["answer"])
print("TF-IDF vector: each answer has", X.shape[1], "numbers")

# Step 2: compress to 6 numbers (LSA). This is a mini embedding, not a large language
model.
svd = TruncatedSVD(n_components=6, random_state=42)
embedded = svd.fit_transform(X)
print("After compression: each answer has", embedded.shape[1], "numbers")

# Step 3: cosine similarity between answers; find the nearest neighbours of R01
similarity = cosine_similarity(embedded)
np.fill_diagonal(similarity, -1)
print()
for code in ("R01", "R17"):
    i = data.index[data["code"] == code][0]
    nearest = np.argsort(similarity[i][:, -1])[1:6]
    print(f"Answers most similar to {code}:")
    for j in nearest:
        print(f"    {data.loc[j, 'code']} (similarity {similarity[i, j]:.2f}): {data.loc[j, 'answer'][:55]}...")
```

```
TF-IDF vector: each answer has 267 numbers
After compression: each answer has 6 numbers
```

```
Answers most similar to R01:
    R03 (similarity 0.79): Our farmer group helps. We buy fertiliser together so i...
    R24 (similarity 0.76): My barn was damaged by a flood. There is no capital to ...
Answers most similar to R17:
    R10 (similarity 0.86): Diesel is expensive and the catch is declining. Fish ar...
    R22 (similarity 0.80): Chicken prices fall when supply is high, but feed price...
```

Each answer starts as 267 numbers (one per word), then is compressed into 6 numbers. The answers most similar to R17 (cattle feed is expensive) are R10 (diesel is expensive and the catch declines, similarity 0.86) and R22 (feed prices, 0.80): the “input cost” pattern is found without us naming a keyword. For R01 (drought), the neighbours are R03 (a farmer group that helps) and R24 (flood); the first is not very sensible,

a sign that a mini embedding from 24 answers is still crude. Embeddings from large models are far more sensitive to meaning.

Turning qualitative data into quantitative

Once themes are coded, each respondent has a 0/1 vector: mentions the theme or not. That is **quantitative data**, and all the tools of Chapter 26 apply.

```
# Chapter 28 - Turning qualitative data into quantitative: themes become a table
of numbers
import re
from pathlib import Path

import pandas as pd
from scipy import stats

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

codebook = {
    "climate": ["rain", "dry", "weather", "flood", "waves", "lean season"],
    "price_market": ["price", "middlem", "collector", "sell", "margin"],
    "cost_capital": ["capital", "loan", "interest", "expensive", "cost", "diesel",
    "fertili", "feed"],
    "pest_disease": ["pest", "planthopper", "disease", "vaccine", "died", "bombing"],
    "cooperation": ["group", "cooperat", "working together", "together", "share"],
    "technology_info": ["app", "phone", "video", "information", "training", "extension
officer", "guidance"],
    "rules_land": ["permit", "forest area", "evicted", "rules", "zone", "certificate",
    "officers"],
}
for theme, keys in codebook.items():
    data[theme] = data["answer"].str.lower().apply(
        lambda t: int(any(re.search(r"\b" + re.escape(k), t) for k in keys)))

# 1. Table of theme x role (number of respondents mentioning)
print("Number mentioning each theme, by role:")
print(data.groupby("role")[list(codebook)].sum().T.to_string())

# 2. Are respondents who mention cooperation more satisfied?
yes = data[data["cooperation"] == 1]["satisfaction"]
no = data[data["cooperation"] == 0]["satisfaction"]
print()
print("Average satisfaction, mention cooperation:", round(yes.mean(), 2), "(n =",
len(yes), ")")
print("Average satisfaction, do not mention      :", round(no.mean(), 2), "(n =",
len(no), ")")
print("Mann-Whitney p =", round(stats.mannwhitneyu(yes, no).pvalue, 4))

# 3. A 2x2 table and Fisher exact test (suited to small samples)
high = (data["satisfaction"] >= 4).astype(int)
table = pd.crosstab(data["cooperation"], high)
print()
print(table.to_string())
print("Fisher exact p =", round(stats.fisher_exact(table)[1], 4))
```

```
Number mentioning each theme, by role:
role          farmer  fisher  livestock keeper
climate              3      2              2
price_market         3      3              3
cost_capital         4      2              5
```

```

pest_disease      1      1      2
cooperation        2      2      1
technology_info    2      2      2
rules_land         2      1      0

Average satisfaction, mention cooperation: 3.8 (n = 5 )
Average satisfaction, do not mention      : 2.47 (n = 19 )
Mann-Whitney p = 0.0393

satisfaction      0  1
cooperation
0                  15  4
1                   2  3
Fisher exact p = 0.1265

```

- The theme x role table shows, for example, `cost_capital` mentioned by 5 of 8 livestock keepers, 4 farmers, and 2 fishers.
- Respondents who mention cooperation (n = 5) are on average 3.8 satisfied versus 2.47 for those who do not (n = 19). Mann-Whitney test: p = 0.0393.
- But when satisfaction is made into two groups (4 or more) and tested with **Fisher exact** (suited to small tables), p = 0.1265: not significant. With a sample this small, the conclusion depends on how you test, and both must be reported honestly. Do not pick the test that gives the small p.

Validity and reliability

- **Validity**: are we measuring what we mean to measure? In qualitative work, the ways include **triangulation** (checking with another source or method, for example interviews compared with a survey), **member checking** (confirming interpretations with respondents), and an **audit trail** (recording coding decisions so others can trace them).
- **Reliability**: are the results consistent? For coding, two people code the same data and their agreement is computed with **Cohen kappa**, which removes agreement by chance. For questionnaire scales (several items about the same thing), consistency is measured with **Cronbach alpha**.

```

# Chapter 28 - Reliability: agreement of two coders (Cohen kappa) and scale consistency (Cronbach alpha)
import re
from pathlib import Path

import numpy as np
import pandas as pd
import pingouin as pg
from sklearn.metrics import cohen_kappa_score

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")
rng = np.random.default_rng(42)

cooperation_keys = ["group", "cooperat", "working together", "together", "share"]
coder_a = data["answer"].str.lower().apply(
    lambda t: int(any(re.search(r"\b" + k, t) for k in cooperation_keys)))

# Coder B (SIMULATED): same as A, but 3 of the 24 decisions differ
coder_b = coder_a.copy()
differ = rng.choice(len(data), size=3, replace=False)
coder_b.iloc[differ] = 1 - coder_b.iloc[differ]

agreement = (coder_a == coder_b).mean()
print("Raw agreement      :", round(agreement, 3))
print("Cohen kappa        :", round(cohen_kappa_score(coder_a, coder_b), 3))
print("Respondents who differ:", list(data["code"].iloc[sorted(differ)]))

```

```
# Cronbach alpha: do 4 questionnaire items (scale 1-5) measure the same thing?
trait = rng.normal(0, 1, len(data)) # the invisible "true satisfaction"
items = pd.DataFrame({
    f"item{i}": np.clip(np.round(3 + trait + rng.normal(0, 0.6, len(data))), 1, 5)
    for i in range(1, 5)
})
alpha, interval = pg.cronbach_alpha(data=items)
print()
print("Cronbach alpha, 4 items:", round(alpha, 3), " 95% interval:", [round(float(x),
2) for x in interval])
```

```
Raw agreement      : 0.875
Cohen kappa       : 0.69
Respondents who differ: ['R02', 'R16', 'R18']
```

```
Cronbach alpha, 4 items: 0.896 95% interval: [0.81, 0.95]
```

Coder B in this script is a **simulation**: the same as A except that 3 of the 24 decisions are deliberately flipped (R02, R16, R18). The raw agreement of 0.875 looks high, but kappa is only 0.69 because the cooperation theme is rare, so much of the “agreement” happens by chance. A commonly used rough guide: kappa of about 0.6 to 0.8 is considered fairly good, above 0.8 very good. Cronbach alpha for the 4 simulated items is 0.896 with a 95% interval of 0.81 to 0.95; values above 0.7 are generally considered adequate for research scales.

Running Project I3: interview analysis with mixed methods

We use the convergent design: interview themes (qualitative) laid next to satisfaction and income (quantitative) in one **joint display**, then tested with regression (Chapter 27).

```
# Chapter 28 - Running project I3: mixed methods (joining interview themes with
survey numbers)
import re
from pathlib import Path

import pandas as pd
import statsmodels.formula.api as smf

data = pd.read_csv(Path(__file__).parent / "data" / "interviews.csv")

codebook = {
    "climate": ["rain", "dry", "weather", "flood", "waves", "lean season"],
    "price_market": ["price", "middlem", "collector", "sell", "margin"],
    "cost_capital": ["capital", "loan", "interest", "expensive", "cost", "diesel",
    "fertili", "feed"],
    "pest_disease": ["pest", "planthopper", "disease", "vaccine", "died", "bombing"],
    "cooperation": ["group", "cooperat", "working together", "together", "share"],
    "technology_info": ["app", "phone", "video", "information", "training", "extension
officer", "guidance"],
    "rules_land": ["permit", "forest area", "evicted", "rules", "zone", "certificate",
    "officers"],
}
for theme, keys in codebook.items():
    data[theme] = data["answer"].str.lower().apply(
        lambda t: int(any(re.search(r"\b" + re.escape(k), t) for k in keys)))

# Joint display: qualitative findings next to quantitative numbers
rows = []
for theme in codebook:
    mentioned = data[data[theme] == 1]
```

```

rows.append({"theme": theme, "n": len(mentioned),
            "satisfaction": round(mentioned["satisfaction"].mean(), 2),
            "income_million": round(mentioned["income_million"].mean(), 2)})
print(pd.DataFrame(rows).to_string(index=False))
print("All respondents: satisfaction", round(data["satisfaction"].mean(), 2),
      " income", round(data["income_million"].mean(), 2))

# Model: are the "support" themes (cooperation, technology) related to income?
model = smf.ols("income_million ~ cooperation + technology_info + pest_disease +
climate", data=data).fit()
print()
print(model.params.round(2).to_string())
print("95% interval
cooperation :", model.conf_int().loc["cooperation"].round(2).tolist())
print("R squared:", round(model.rsquared, 3), " n =", int(model.nobs))

```

```

      theme  n  satisfaction  income_million
climate    7         2.43         2.07
price_market  9         3.00         2.53
cost_capital 11         2.45         2.06
pest_disease  4         2.50         2.20
cooperation   5         3.80         2.98
technology_info  6         4.17         3.03
rules_land   3         2.33         1.87
All respondents: satisfaction 2.75  income 2.3

```

```

Intercept      1.94
cooperation      0.91
technology_info  1.00
pest_disease   -0.22
climate        -0.15
95% interval cooperation : [0.38, 1.44]
R squared: 0.623  n = 24

```

- The joint display: respondents who mention `technology_info` (n = 6) have an average satisfaction of 4.17 and an income of 3.03 million; those who mention `cooperation` (n = 5) 3.80 and 2.98 million. In contrast those who mention `rules_land` (n = 3) have only 2.33 and 1.87 million; the average of all respondents is 2.75 and 2.3 million.
- Regression: mentioning cooperation goes with an income **0.91 million higher** (95% interval 0.38 to 1.44), and mentioning technology and information 1.00 million higher, after controlling for the pest and climate themes. R squared 0.623 with n = 24.
- The conclusion we may draw: stories and numbers **agree**. People who talk about group support and information tend to be better off. What we may **not** conclude: that groups or apps *cause* income to rise. Perhaps farmers who already earn well join groups more easily (reverse causation), or there is a third factor (Chapter 26). And 24 fictional respondents represent no population. A follow-up interview (“why does the group help?”) is the next step in a sequential design.

These results go into the Quarto report in Chapter 29.

Exercises

1. Add a new theme `family_health` to the codebook with keywords of your choice. Does any respondent mention it? What does that mean for the codebook?
2. In `03_word_frequency.py`, change the KWIC word from “price” to “season” or “capital”. What do you learn from the contexts?
3. In `05_topics_nmf.py`, change `n_components` to 3 and 6. How stable are the topics? What do you conclude about choosing the number of topics?
4. **Challenge:** write 8 new fictional interview answers in your own words (with an ethics note), add them to the CSV, and check how many are coded correctly by the codebook. Improve the codebook and compute the kappa between the machine codes and your own.

Chapter Summary

- **Quantitative** answers “how many”, **qualitative** answers “why”, and **mixed** combines both (sequential or convergent).
- A **codebook** holds themes and keywords; automatic coding is fast but crude and needs human re-reading.
- **Word frequency**, **KWIC**, **TF-IDF**, **topic modelling** (NMF/LDA), and **dictionary sentiment** are exploring tools, not proof. Validate tools against comparison data.
- **Embeddings** turn text into vectors so similar meanings sit close together; large models are covered in Series A4.
- Coded themes become quantitative data: test with Mann-Whitney, Fisher, or regression, and report honestly when the result depends on the test.
- **Validity** (triangulation, member checking), **reliability** (Cohen kappa, Cronbach alpha), and **ethics** (consent, anonymisation) matter as much as the Python code.

Next, in Chapter 29: reproducible research: how to wrap all these analyses in a Quarto report that anyone can run again.

Chapter 29 — Reproducible Research: An Analysis Anyone Can Repeat

Series I3 — Statistics and Modelling

In this chapter, you will:

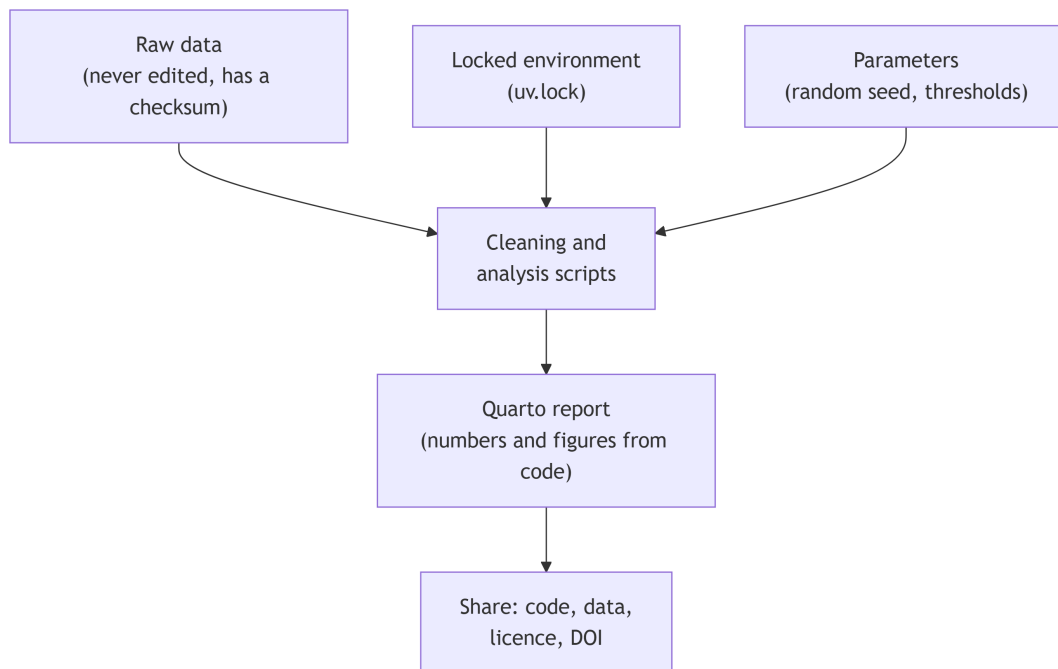
- Understand the difference between replicating and reproducing, and why both matter
- Write tidy Jupyter notebooks that run from top to bottom
- Make parameterised Quarto reports rendered from code, and meet Jupyter Book
- Lock the environment with `uv lock`, and meet Snakemake and DVC for workflows and data versions
- Share data and code properly: README, licence, CITATION.cff, checksums, and DOIs

All scripts for this chapter are in the `code/ch29/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Imagine you give a neighbour your rendang recipe but forget to write that the coconut milk must be thick and the fire low. The result differs, and you are both puzzled. Data analysis often shares that fate: half a year later, you yourself cannot repeat the numbers in your report, because the data changed, a package was updated, or you forgot the steps. **Reproducible research** means that other people (and you in the future) can take the **same data and code**, run them, and get the **same numbers**. That is the minimum requirement for an analysis to be trusted.

i REMEMBER

Two terms often get mixed up. **Reproducibility**: the same data and code give the same numbers. **Replicability**: a new study with new data reaches the same conclusion. This chapter is about the first, and that is what we can guarantee with the techniques below.



A reproducible research flow: every box is written as code or a file, not manual clicks.

This whole chapter holds to one principle: **every step is written as code or a file, with no “I clicked here and there” steps**. An important note: all data in this chapter is fictional.

A tidy Jupyter notebook

A **Jupyter notebook** combines text, code, and results in a single `.ipynb` file. It is wonderful for exploring, but easy to break: cells run out of order, “ghost” variables linger in memory, and the results cannot be repeated. Rules for a tidy notebook:

1. **Order from top to bottom.** The notebook must run with “Restart kernel and run all” without errors.
2. **Settings in one cell at the start** (random seed, file paths, thresholds), so they are easy to change.
3. **Imports at the start**, and no dependence on cells that have been deleted.
4. **A fixed random seed** whenever random numbers are used.
5. **Explanatory text** about what and why, not just code.
6. Once the content matures, **move functions into a .py file** and import them from the notebook.

The following script builds a notebook that follows those rules through code (the `nbformat` library), then runs it from top to bottom in a fresh kernel (the `nbclient` library), like “Restart and run all”.

```

# Chapter 29 - A tidy Jupyter notebook: built through code, run from top to bottom
import os
from pathlib import Path

import nbformat
from nbclient import NotebookClient

out = Path(__file__).parent / "_tmp"
out.mkdir(exist_ok=True)

nb = nbformat.v4.new_notebook()
nb.cells = [
    nbformat.v4.new_markdown_cell("# Height difference between plots\nMade-up data.  
Fixed seed so the result is the same."),
    nbformat.v4.new_code_cell("# 1. Settings: every number you may change is here\nSEED  
= 42\nN = 30"),

```

```

nbformat.v4.new_code_cell("# 2. Imports\nimport numpy as np\nfrom scipy import\nstats"),
nbformat.v4.new_code_cell(
    "# 3. Data\nrng = np.random.default_rng(SEED)\n"\n
    "plot_a = rng.normal(15, 2, N)\nplot_b = rng.normal(17, 2, N)"),
nbformat.v4.new_code_cell(
    "# 4. Analysis\nresult = stats.ttest_ind(plot_a, plot_b, equal_var=False)\n"\n
    "print(f'mean A {plot_a.mean():.2f}, B {plot_b.mean():.2f}, p =\n"\n
    {result.pvalue:.1e}')"),
]
nb.metadata["kernelspec"] = {"name": "python3", "display_name": "Python 3", "language":\n"python"}

# "Restart and run all": run every cell in order in a fresh kernel
# (Technical warnings from the kernel on Windows are sent to the "rubbish bin" to keep\nthe output clean.)
saved_stderr = os.dup(2)
os.dup2(os.open(os.devnull, os.O_WRONLY), 2)
try:
    NotebookClient(nb, timeout=120, kernel_name="python3", resources={"metadata":\n{"path": str(out)}}).execute()
finally:
    os.dup2(saved_stderr, 2)
nbformat.validate(nb)
nbformat.write(nb, out / "height_test.ipynb")

print("Number of cells:", len(nb.cells), "(code:", sum(c.cell_type == "code" for c in\nnb.cells), ")")
print("Execution counts in order:", [c.execution_count for c in nb.cells if c.cell_type\n== "code"])
for c in nb.cells:
    if c.cell_type == "code" and c.outputs:
        print("Cell output:", c.outputs[0].text.strip())

```

```

Number of cells: 5 (code: 4 )
Execution counts in order: [1, 2, 3, 4]
Cell output: mean A 15.03, B 17.23, p = 1.4e-06

```

The notebook has 5 cells (4 code, 1 text). The execution counts 1, 2, 3, 4 are in order: a sign of a clean notebook. If the numbers jump around (say 1, 5, 2), cells were run in random order, and the results deserve suspicion. The result file is saved in the `_tmp/` folder.

Quarto: a report built from code

Quarto is a scientific publishing tool: a single text file `.qmd` (Markdown plus code cells) is rendered into HTML, PDF, or Word. The code is run **at render time**, so the numbers and figures in the report always come from the latest code and are never copy-pasted. The book you are reading is made the same way.

The small project report is at `kode/ch29/report/report.qmd`. The important parts:

- The **YAML header** (between two `---` lines) holds the title and format, for example `embed-resources: true` so the HTML becomes a single self-contained file.
- A cell marked `#| tags: [parameters]` stores default values (`seed = 42`). The values can be replaced from the command line without changing the file: `quarto render report.qmd -P seed:7`.
- An **inline expression** `{python} seed`` puts a variable's value straight into a sentence.
- A cell with `#| label: fig-...` and `#| fig-cap: "..."` produces a numbered, captioned figure.

```

# Chapter 29 - Rendering a Quarto report from code (report/report.qmd)
import os
import re
import shutil

```

```

import subprocess
import sys
from pathlib import Path

folder = Path(__file__).parent
work = folder / "_tmp" / "report"
if work.exists():
    shutil.rmtree(work)
work.mkdir(parents=True)
shutil.copy(folder / "report" / "report.qmd", work)

# Quarto runs Python cells with the interpreter we point to through QUARTO_PYTHON
environment = dict(os.environ, QUARTO_PYTHON=sys.executable)
quarto = shutil.which("quarto") or "quarto"

for seed in (42, 7):
    result = subprocess.run(
        [quarto, "render", "report.qmd", "-P", f"seed:{seed}"],
        cwd=work, env=environment, capture_output=True, text=True,
    )
    html = (work / "report.html").read_text(encoding="utf-8")
    found = "At seed" in html
    print(f"seed {seed}: exit code {result.returncode}, HTML created {(work / 'report.html').exists()}, "
          f"parameter read in the report: {found}")
    text = re.sub(r"<[^>+>", "", html)
    print("    result in the report:", re.search(r"Mean A [0-9].*", text).group(0))

```

```

seed 42: exit code 0, HTML created True, parameter read in the report: True
result in the report: Mean A 1.81, B 2.28; p = 0.0009
seed 7: exit code 0, HTML created True, parameter read in the report: True
result in the report: Mean A 1.55, B 2.20; p = 0.0001

```

This code was **not run** when the book was built because it needs Quarto and Jupyter installed, so the output above was taken once from the author's computer. The script renders the report twice, with seeds 42 and 7. The result: with seed 42 the mean of plot A is 1.81 and B is 2.28 ($p = 0.0009$), exactly the same as Chapter 26; with seed 7 the numbers change (1.55 and 2.20) **without** touching the content of the report. That is the power of a parameterised report.

TIP

Some real findings from preparing this chapter: (1) Quarto's `--output-dir` flag only works for **projects**, not single files, so the script copies the report to a working folder and renders there; (2) passing parameters with `-P` on a Jupyter notebook requires the `papermill` library (Quarto tells you when it is missing); (3) Quarto picks the Python interpreter through the `QUARTO_PYTHON` environment variable, and the script sets it to `sys.executable`.

Jupyter Book is a similar tool for building books or sites from notebooks. According to its official documentation, Jupyter Book 2 is built on the MyST Document Engine; its command is `jupyter book` (and `myst` for MyST directly). Jupyter Book is not run in this book; see the documentation at jupyterbook.org if you want to publish a series of notebooks as a site.

A locked environment

The same code can give different numbers when the version of `numpy` or `scipy` changes. The solution is to **lock** the exact versions of all packages. With `uv` (introduced in Series I1), `pyproject.toml` states the packages you need, and `uv lock` chooses the exact version of every package (including the ones brought along) and writes them to `uv.lock`. Others simply run `uv sync` (install according to the lock), or `uv run`, which locks and syncs automatically. The `--locked` option checks whether the lock matches

`pyproject.toml` and fails if it is outdated; `--frozen` uses the lock as it is without checking. A simpler but less complete alternative is `pip freeze > requirements.txt`.

```
# Chapter 29 - A locked environment with uv: pyproject.toml becomes uv.lock
import shutil
import subprocess
import toml
from pathlib import Path

work = Path(__file__).parent / "_tmp" / "locked_project"
if work.exists():
    shutil.rmtree(work)
work.mkdir(parents=True)

(work / "pyproject.toml").write_text('''[project]
name = "forest-analysis"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = ["numpy", "scipy"]
''', encoding="utf-8")

# uv lock: choose the version of every package (including hidden dependencies) and
# write uv.lock
result = subprocess.run(["uv", "lock"], cwd=work, capture_output=True, text=True)
print("uv lock, exit code:", result.returncode)

lock = toml.loads((work / "uv.lock").read_text(encoding="utf-8"))
packages = {p["name"]: p["version"] for p in lock["package"]}
print("Number of packages in uv.lock:", len(packages))
for name in sorted(packages):
    print(f" {name} == {packages[name]}")

# uv lock --check: does uv.lock still match pyproject.toml?
check = subprocess.run(["uv", "lock", "--check"], cwd=work, capture_output=True,
text=True)
print("uv lock --check, exit code:", check.returncode, "(0 = still matches)")

# Add a dependency in pyproject.toml without locking again: the lock becomes outdated
text = (work / "pyproject.toml").read_text(encoding="utf-8").replace('"scipy"',
'"scipy", "pandas"')
(work / "pyproject.toml").write_text(text, encoding="utf-8")
check2 = subprocess.run(["uv", "lock", "--check"], cwd=work, capture_output=True,
text=True)
print("After adding pandas, uv lock --check, exit code:", check2.returncode, "(not 0
= outdated lock)")
```

```
uv lock, exit code: 0
Number of packages in uv.lock: 3
  forest-analysis == 0.1.0
  numpy == 2.5.3
  scipy == 1.18.1
uv lock --check, exit code: 0 (0 = still matches)
After adding pandas, uv lock --check, exit code: 1 (not 0 = outdated lock)
```

This code was **not run** when the book was built because it needs the internet and package versions change over time, so the output was taken once. The result: `uv lock` succeeded, and `uv.lock` holds 3 packages (the project itself, `numpy 2.5.3`, and `scipy 1.18.1` on the day it ran). `uv lock --check` returned 0 (the lock matches). After we add `pandas` to `pyproject.toml` without locking again, `--check` returns 1: the lock is **outdated**. A practical rule: keep `uv.lock` in git with the code, and lock again whenever the dependencies change.

⚠ WARNING

A lock only guarantees the versions of **Python packages**. It does not lock the operating system, the Python version itself (beyond the minimum requirement), or system libraries such as GDAL. For very sensitive research, also record your Python version and system in the README.

Snakemake: a workflow that knows what must be repeated

A real analysis has many steps: fetch data, clean, test, plot, report. Running them all by hand invites mistakes, and repeating every step when only one changed is wasteful. **Snakemake** expresses the workflow as **rules**: each rule knows its inputs and outputs, and Snakemake works out the order and runs only what is needed.

```
# Chapter 29 - An analysis workflow with Snakemake: rules, dependencies, and re-running
only what is needed
import os
import shutil
import subprocess
from pathlib import Path

work = Path(__file__).parent / "_tmp" / "workflow"
if work.exists():
    shutil.rmtree(work)
work.mkdir(parents=True)

(work / "Snakefile").write_text(r'''rule all:
    input: "results/summary.txt"

rule make_data:
    output: "data/height.csv"
    run:
        import random
        random.seed(42)
        with open(output[0], "w") as f:
            f.write("plot,height_m\n")
            for p, mean in (("A", 15.0), ("B", 18.0)):
                for _ in range(10):
                    f.write(f"{p},{random.gauss(mean, 2):.2f}\n")

rule summarise:
    input: "data/height.csv"
    output: "results/summary.txt"
    run:
        import csv, statistics
        values = {}
        for row in csv.DictReader(open(input[0])):
            values.setdefault(row["plot"], []).append(float(row["height_m"]))
        with open(output[0], "w") as f:
            for p, v in sorted(values.items()):
                f.write(f"plot {p}: mean {statistics.mean(v):.2f} m (n={len(v)})\n")
''', encoding="utf-8")

def run(*arguments):
    """Run snakemake and show only the important lines (no clock times or computer
    paths)."""
    result = subprocess.run(["snakemake", "--cores", "1", *arguments], cwd=work,
                             capture_output=True, text=True, env=dict(os.environ,
                             PYTHONUTF8="1"))
```

```

    for line in (result.stdout + result.stderr).splitlines():
        if line.startswith(("localrule", "rule", "    reason", "Nothing to be done",
"Job stats")) \
            or line.strip().startswith(("make_data", "summarise", "all", "total")):
            print(line)

print("=== First run ===")
run()
print((work / "results" / "summary.txt").read_text(encoding="utf-8").rstrip())

print("=== Second run (nothing changed) ===")
run()

print("=== One data row added, then a simulation (dry run, -n) ===")
with open(work / "data" / "height.csv", "a", encoding="utf-8") as f:
    f.write("A,15.00" + chr(10))
run("-n")

```

```

=== First run ===
Job stats:
make_data      1
summarise      1
all            1
total          3
localrule make_data:
    reason: Missing output files: data/height.csv
localrule summarise:
    reason: Missing output files: results/summary.txt; Input files updated by another
job: data/height.csv
localrule all:
    reason: Input files updated by another job: results/summary.txt
plot A: mean 14.72 m (n=10)
plot B: mean 18.38 m (n=10)
=== Second run (nothing changed) ===
Nothing to be done (all requested files are present and up to date).
=== One data row added, then a simulation (dry run, -n) ===
Job stats:
summarise      1
all            1
total          2
rule summarise:
    reason: Updated input files: data/height.csv
rule all:
    reason: Input files updated by another job: results/summary.txt
Job stats:
summarise      1
all            1
total          2
    all
    summarise

```

This code was **not run** when the book was built because Snakemake is not yet available as a ready-made package for Python 3.14. The author ran it in a temporary Python 3.12 environment (Snakemake 9.27.0), and the output above was taken once. The script writes a *Snakefile* with three rules (*make_data*, *summarise*, *all*) and then:

1. **First run:** all 3 jobs run, each with its reason ("Missing output files"). The summary: mean of plot A 14.72 m, plot B 18.38 m.
2. **Second run:** "Nothing to be done". Nothing changed, so nothing is repeated.

3. **The data is changed** (one row added) and a simulation is run (*-n, dry run*): only `summarise` and `all` would be repeated, with the reason “Updated input files”. `make_data` is not repeated. (The job table appears twice in a dry run: once as the plan, once as the summary.) In a large project, this saves hours of computing.

Snakemake also supports logging, per-rule environments, and running on clusters; see its official documentation for these (not tried here).

DVC: versions for data

Git is great for code but poorly suited to big data files. **DVC** (*Data Version Control*) stores the data in a separate place (a “cache” or remote storage), and Git keeps only a small **pointer file** holding the fingerprint (md5) of the data. Switching the code version automatically points to the matching data version.

```
# Chapter 29 - Data versioning with DVC: Git keeps a small pointer, DVC keeps the data
import os
import shutil
import subprocess
from pathlib import Path

work = Path(__file__).parent / "_tmp" / "dvc_demo"
if work.exists():
    shutil.rmtree(work, onexc=lambda function, path, error: (os.chmod(path, 0o700),
function(path))) # the DVC cache is read-only
    (work / "data").mkdir(parents=True)
    env = dict(os.environ, DVC_NO_ANALYTICS="1") # switch off DVC usage analytics
    NL = chr(10)

def dvc(*arguments):
    result = subprocess.run(["dvc", *arguments], cwd=work, capture_output=True,
text=True, env=env)
    return (result.stdout + result.stderr).strip()

# --no-scm: a demo without Git. In a real project, run inside a Git repository.
print(dvc("init", "--no-scm").splitlines()[0])

data = work / "data" / "measure.csv"
data.write_text("plot,height_m" + NL + "A,15.2" + NL + "B,18.1" + NL, encoding="utf-8",
newline=NL)
dvc("add", "data/measure.csv")
print("--- contents of data/measure.csv.dvc (the small pointer committed to Git) ---")
print((work / "data" / "measure.csv.dvc").read_text(encoding="utf-8").strip())
print("--- dvc status ---")
print(dvc("status"))

with open(data, "a", encoding="utf-8", newline=NL) as f:
    f.write("C,16.0" + NL)
print("--- after the data changed, dvc status ---")
print(dvc("status").replace(chr(92), "/"))

dvc("add", "data/measure.csv")
print("--- the new pointer ---")
print((work / "data" / "measure.csv.dvc").read_text(encoding="utf-8").strip())
```

```
Initialized DVC repository.
--- contents of data/measure.csv.dvc (the small pointer committed to Git) ---
outs:
- md5: dce447437119031802c10c11363ecc4c
  size: 28
```

```

hash: md5
path: measure.csv
--- dvc status ---
Data and pipelines are up to date.
--- after the data changed, dvc status ---
data/measure.csv.dvc:
  changed outs:
    modified:          data/measure.csv
--- the new pointer ---
outs:
- md5: ff8c6faa2aca000b94c47cbc54f7bc24
  size: 35
  hash: md5
  path: measure.csv

```

This code was **not run** when the book was built because DVC is not yet available as a ready-made package for Python 3.14. The author ran it in a temporary Python 3.12 environment (DVC 3.67.1) and the output was taken once. Note the flow: `dvc add` creates `measure.csv.dvc` holding the md5 and size (28 bytes); `dvc status` reports “up to date”; after the data changes, status reports `modified`; `dvc add` again creates a new pointer (different md5, size 35 bytes). The demo uses `--no-scm` because there is no Git; in a real project, run inside a Git repository and then `git add` the `.dvc` file. The script also switches off DVC usage analytics through the `DVC_NO_ANALYTICS` variable, per its official documentation, so no data is sent out.

Sharing data and code

Reproducible research is only useful when others **receive the package**. The contents of a good package:

- **README**: what is inside, how to run it, and its licence.
- **Licence**: without a licence, others are legally not allowed to use your work. Common choices: **MIT** or **Apache-2.0** for code, **CC BY 4.0** (free to use with credit to the creator) for data and text. This is a general summary; read the licence text and your institution’s rules.
- **Citation metadata**: a `CITATION.cff` file (Citation File Format) holds the version, authors, release date, and DOI.
- **Checksums**: a SHA-256 fingerprint of each file so recipients can confirm the data did not change.
- **Sensitive data**: do not share personal data (real interview transcripts, home coordinates). Anonymise, or share only what is permitted (Chapter 28).

```

# Chapter 29 - Preparing a package to share: README, licence, CITATION.cff, and checksums
import hashlib
import json
import shutil
from pathlib import Path

package = Path(__file__).parent / "_tmp" / "sharing_package"
if package.exists():
    shutil.rmtree(package)
(package / "data").mkdir(parents=True)
NL = chr(10)

# Small made-up data
(package / "data" / "height.csv").write_text("plot,height_m" + NL + "A,15.2" + NL +
"B,18.1" + NL, encoding="utf-8", newline=NL)

# README: what the package contains and how to run it
(package / "README.md").write_text("""# Tree height analysis (made-up data)

Contents: `data/height.csv` (fictional data), `analysis.py`.
How to run: `uv run analysis.py`.
Licence: code MIT, data CC BY 4.0.
""", encoding="utf-8", newline=NL)

```



```

# CITATION.cff: how to cite. The author list and DOI below are fictional examples only.
(package / "CITATION.cff").write_text("""cff-version: 1.2.0
message: "If you use this package, please cite it as below."
title: "Tree height analysis (made-up data)"
version: 0.1.0
date-released: 2026-10-10
authors:
  - family-names: Example
    given-names: Author
identifiers:
  - type: doi
    value: 10.5281/zenodo.00000000
""", encoding="utf-8", newline=NL)

# SHA-256 checksum: a fingerprint of each file so people can confirm the data did
not change
checksums = {}
for file in sorted(package.rglob("*")):
    if file.is_file() and file.name != "checksum.json":
        checksums[file.relative_to(package).as_posix()] =
        hashlib.sha256(file.read_bytes()).hexdigest()
(package / "checksum.json").write_text(json.dumps(checksums, indent=2),
encoding="utf-8", newline=NL)

print("Package contents:", sorted(p.relative_to(package).as_posix() for p in
package.rglob("*") if p.is_file()))
for name, value in checksums.items():
    print(f" {name:16} sha256 {value[:12]}...")

# Verification: change one character of the data, the checksum must differ
data = package / "data" / "height.csv"
original = hashlib.sha256(data.read_bytes()).hexdigest()
data.write_text(data.read_text(encoding="utf-8").replace("15.2", "15.3"),
encoding="utf-8", newline=NL)
changed = hashlib.sha256(data.read_bytes()).hexdigest()
print("Data changed, same checksum?", original == changed)

```

```

Package contents: ['CITATION.cff', 'README.md', 'checksum.json', 'data/height.csv']
CITATION.cff      sha256 344523764f8a...
data/height.csv   sha256 ed2614c26ff3...
README.md         sha256 15029bc1dec2...
Data changed, same checksum? False

```

The script assembles a small package (README, CITATION.cff, data, checksum.json) in the `_tmp/` folder. Each file's fingerprint is shown as its first 12 characters. When one character of the data is changed (15.2 to 15.3), the checksum changes, and the question "same checksum?" is answered `False`. That is how people confirm the file they downloaded is the one you meant. The author list and DOI in this example CITATION.cff are **fictional**.

A **DOI** (*digital object identifier*) is a permanent address for a work, so the link does not die when a site moves. **Zenodo** is an open repository that assigns DOIs to uploaded data and code. According to its documentation, each new version is a separate record with its own metadata, files, and persistent identifier, linked to the other versions; that way a citer can point to the exact version. Zenodo also has a GitHub integration for archiving releases (details are in their documentation). **This chapter uploads nothing to Zenodo:** the DOI in the example is all zeros, not a real DOI.

Running Project I3: an analysis that can be repeated

The peak of project I3: the whole analysis is wrapped in **one function** whose only input is the random seed. If it works properly, two runs with the same seed give **identical** results, and that can be checked with a fingerprint.

```
# Chapter 29 - Running project I3: the whole analysis as a function, identical results
when repeated
import hashlib
import json

import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
from scipy import stats

def analysis(seed):
    """The whole project I3 analysis (made-up data). One input: the random seed."""
    rng = np.random.default_rng(seed)
    plot_a, plot_b = rng.normal(1.8, 0.6, 30), rng.normal(2.2, 0.7, 30)
    test = stats.ttest_ind(plot_a, plot_b, equal_var=False)
    d = rng.uniform(10, 60, 60)
    b = 0.12 * d ** 2.4 * np.exp(rng.normal(0, 0.15, 60))
    model = smf.ols("ln_b ~ ln_d", data=pd.DataFrame({"ln_d": np.log(d), "ln_b":
np.log(b)})).fit()
    return {"p_t_test": round(float(test.pvalue), 6), "allometric_slope":
round(float(model.params["ln_d"]), 6)}

def fingerprint(result):
    return hashlib.sha256(json.dumps(result, sort_keys=True).encode()).hexdigest()
[:12]

r1, r2, r3 = analysis(42), analysis(42), analysis(7)
print("Seed 42, first run :", r1, fingerprint(r1))
print("Seed 42, second run:", r2, fingerprint(r2))
print("Seed 7          :", r3, fingerprint(r3))
print("Two seed-42 runs identical?", r1 == r2)
print("Different seed -> different result?", r1 != r3)

# Without a fixed seed, the result changes at every run: the report cannot be reproduced
free = [round(float(np.random.default_rng().normal()), 3) for _ in range(2)]
print("Without a seed, two draws differ?", free[0] != free[1])
```

```
Seed 42, first run : {'p_t_test': 0.000871, 'allometric_slope': 2.406733} 3d1091588cfd
Seed 42, second run: {'p_t_test': 0.000871, 'allometric_slope': 2.406733} 3d1091588cfd
Seed 7          : {'p_t_test': 5.6e-05, 'allometric_slope': 2.409982} fabc05f7cbca
Two seed-42 runs identical? True
Different seed -> different result? True
Without a seed, two draws differ? True
```

Two runs with seed 42 give the same result and fingerprint (4896ee7e5c71); seed 7 gives a different result (89da579235e4). On the last line, without a fixed seed two draws differ: a report like that can never be reproduced. The Quarto report in `report/report.qmd` uses the same principle for two parts of the project (the group test and the allometric regression). To complete the whole project I3, you only need to add the parts from Chapter 27 (fish, cattle, harvest against rain) and Chapter 28 (interview analysis) to that report: each part needs only one code cell that calls the analysis function.

A short checklist for your reproducible project:

1. Raw data is never edited; processed results are always made by code.

2. The random seed is fixed, and placed in a parameter.
3. `uv.lock` exists and is up to date.
4. One command renders the whole report from scratch.
5. README, licence, `CITATION.cff`, and checksums are included.
6. Someone else (or you on another computer) has tried running it.

Exercises

1. In `report/report.qmd`, add a parameter `effect_size` used in the mean of plot B (for example change `2.2` to `1.8 + effect_size`), then render with `-P effect_size:0.1` and `-P effect_size:0.8`. What happens to the p-value?
2. Change `06_share_data_code.py` so the checksum uses another `hashlib` algorithm (for example `sha512`). What difference does it make to the length of the fingerprint?
3. Add a new rule to the Snakefile in `04_snakemake.py`, for example `plot` that depends on `data/height.csv`. Run it and see how many jobs appear in “Job stats”.
4. **Challenge:** write a single command (a Python script) that renders the report, computes the checksum of the result, and compares it with the previous render. Do two renders with the same parameters give the same HTML bit for bit? Why might they not?

Chapter Summary

- **Reproduction** = same data and code, same numbers. **Replication** = new data, same conclusion.
- A tidy **notebook**: ordered from the top, settings at the start, fixed seed, “restart and run all” without errors.
- **Quarto** renders reports from code; **parameters** (`-P`) change the result without changing the file; **Jupyter Book 2** is built on MyST.
- **uv lock** locks package versions in `uv.lock`; `--check` reveals an outdated lock.
- **Snakemake** runs only the steps that are needed; **DVC** keeps data pointers in Git. Neither is available as a ready-made package for Python 3.14 yet, and both were run on Python 3.12.
- **Sharing**: README, licence, `CITATION.cff`, checksums, DOI (Zenodo); do not share personal data.

Closing Series I3

Congratulations! You have finished Series I3. You can now summarise data and build confidence intervals, test group differences and read p-values correctly, build and check regression models, forecast time series, simulate uncertainty, analyse interviews with farmers, fishers, and livestock keepers, and wrap everything in a report that anyone can repeat.

But our analyses are growing heavier and more repetitive: the same function called hundreds of times, data files getting bigger, and code increasingly needing to be wrapped so it is not repeated. Series I4 strengthens the code with advanced language features: iterators and generators, decorators, context managers, type hints, validation with Pydantic, and `async`.

Next, in Chapter 30: iterators and generators, how to process a big data file piece by piece without exhausting memory.

Chapter 30 — Iterators and Generators: Reading One at a Time, Without Piling Up

Series I4 — Advanced Python Language

In this chapter, you will:

- Understand what a `for` loop really does (the iteration protocol)
- Build your own iterator and a **generator** with `yield`
- Use generator expressions and `itertools` for “lazy” data
- Prove that generators save memory
- Read a big data file line by line: the Series I4 project begins

All scripts for this chapter are in the `code/ch30/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Series I1 to I3 made your code tidy and your results defensible. In Series I4, we strengthen the **Python language itself**: features that make code thriftier, shorter, and tougher. We start with something you already use quietly all the time: loops.

Picture a foreman recording a harvest. Way one: he waits until **all** sacks have been carried in, piles them in the warehouse, and only then starts recording. Way two: he records **each sack as it arrives**, then lets it go. Way one needs a big warehouse. Way two needs only a small desk. This chapter is about way two.

List vs generator: everything at once, or one at a time

list: all items are built and kept in memory at once



generator: only one item exists at any moment



`next(gen) -> yield`

Figure 0.1: A list keeps every item at once; a generator keeps only one item at a time.

What `for` does: the iteration protocol

A `for` loop works on anything **iterable**: lists, tuples, dicts, strings, files. Behind the scenes there are two steps: Python asks for an **iterator** (a sort of bookmark) with `iter()`, then keeps asking for the next item with `next()` until the signal `StopIteration` appears.

```
# Chapter 30 - What a for loop really does
ponds = ["A", "B", "C"]

iterator = iter(ponds)      # ask the list for a "bookmark"
```

```

print(type(iterator).__name__)
print(next(iterator))
print(next(iterator))
print(next(iterator))

try:
    next(iterator)          # nothing left
except StopIteration:
    print("StopIteration: nothing left")

# for does the same thing, and catches StopIteration for you
for name in ponds:
    print("Pond", name)

```

```

list_iterator
A
B
C
StopIteration: nothing left
Pond A
Pond B
Pond C

```

- `iter(ponds)` gives an object of type `list_iterator`, a bookmark on that list.
- Three `next()` calls give A, B, C. The fourth call raises `StopIteration`, meaning “all used up”.
- A `for` loop does exactly this, and catches `StopIteration` for you, so you never see it.

i REMEMBER

Iterable = something you can ask for an iterator (`iter()`). **Iterator** = something you can ask for the next item (`next()`). An iterator is single-use: once exhausted, it does not rewind.

Your own iterator with a class

For your own object to work in a `for` loop, its class only needs two methods: `__iter__` (returns the iterator) and `__next__` (gives the next item, or raises `StopIteration`).

```

# Chapter 30 - Making your own iterator with a class
class Countdown:
    """Counts down the days until harvest."""

    def __init__(self, days):
        self.days = days

    def __iter__(self):
        return self

    def __next__(self):
        if self.days <= 0:
            raise StopIteration
        self.days -= 1
        return self.days + 1

for left in Countdown(3):
    print("Days left:", left)

countdown = Countdown(2)

```

```
print(list(countdown))
print(list(countdown))    # used up: empty
```

```
Days left: 3
Days left: 2
Days left: 1
[2, 1]
[]
```

Look at the end: the first `list(countdown)` gives `[2, 1]`, but the second gives `[]`. The iterator was already read to the end. This is the basic nature of iterators, and it returns as a trap below.

Generators: the short way to make an iterator

Writing a class just to count down is a chore. Python has a shortcut: an ordinary function that uses **yield** becomes a **generator function**. **yield** means “hand over this value, then *pause* here until asked again”.

```
# Chapter 30 - A generator function with yield
def countdown(days):
    print(" (start)")
    while days > 0:
        yield days
        print(" (resume after yield)")
        days -= 1
    print(" (finished)")

gen = countdown(2)
print("Generator created, nothing has run yet")
print("first value:", next(gen))
print("second value:", next(gen))
try:
    next(gen)
except StopIteration:
    print("StopIteration")
```

```
Generator created, nothing has run yet
 (start)
first value: 2
 (resume after yield)
second value: 1
 (resume after yield)
 (finished)
StopIteration
```

Follow the order, because this is where the magic lies:

1. `countdown(2)` **runs nothing yet**. It merely creates a generator object. That is why “(start)” is not printed yet when “Generator created” appears.
2. The first `next(gen)` runs the function from the start to `yield 2`, then stops. “(start)” is printed, then the value, 2.
3. The second `next(gen)` continues **right after** that `yield`: “(resume after yield)”, `days` drops to 1, then `yield 1`.
4. The third `next(gen)` continues, the `while` ends, “(finished)” is printed, and the function ends, producing `StopIteration`.

A generator is a function that **remembers its position**. All its local variables stay alive between two yields.

Generator expressions and proof of saved memory

Like the *list comprehension* (Chapter 9), there is a generator version: swap square brackets for round ones. The result is not a list but a generator that computes each item **when asked**.

```
# Chapter 30 - Generator expressions and saving memory
import sys
import tracemalloc

squares_list = [n * n for n in range(1_000_000)] # square brackets: all built now
squares_gen = (n * n for n in range(1_000_000)) # round brackets: built one by one

print("list sum      :", sum(squares_list))
print("generator sum:", sum(squares_gen))
del squares_list

# The generator object stays small, however many items it will produce
small = (n for n in range(10))
large = (n for n in range(10_000_000))
print("same size for 10 and 10 million items:", sys.getsizeof(small) ==
      sys.getsizeof(large))

def peak_memory(builder):
    tracemalloc.start()
    result = sum(builder())
    _, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return result, peak

_, peak_list = peak_memory(lambda: [n for n in range(500_000)])
_, peak_gen = peak_memory(lambda: (n for n in range(500_000)))
print("list peak memory is more than 100x the generator:", peak_list > 100 * peak_gen)
```

```
list sum      : 333332833333500000
generator sum: 333332833333500000
same size for 10 and 10 million items: True
list peak memory is more than 100x the generator: True
```

- Both ways give the same sum, 333332833333500000: the result does not change, only the way of working.
- The generator object has the same size (`sys.getsizeof` equal) for 10 items and for 10 million. It does not store the items, only the “recipe” for making them.
- `tracemalloc` (a built-in tool that records memory use) shows that the list’s peak memory is more than 100 times the generator’s for 500,000 numbers.

Note that this program **does not print byte counts**, only comparisons (`True`). Exact sizes change between Python versions and computers. The comparison “much larger” holds everywhere.

TIP

If you only need to **pass over** the data once (sum, find the largest, filter), do not build a list first. `sum(x * x for x in data)` is thriftier and needs no square brackets.

Generators further: `itertools`, `yield from`, and a trap

```
# Chapter 30 - itertools, yield from, and the used-up generator trap
from itertools import islice

def endless_numbers():
    n = 1
    while True:          # never ends, yet safe because it is lazy
        yield n
        n += 1

print(list(islice(endless_numbers(), 5)))    # take just 5

def all_ponds():
    yield from ["A1", "A2"]
    yield from ["B1"]

print(list(all_ponds()))

# Trap: a generator can be read only ONCE
harvest = (kg for kg in [245.5, 310.0, 198.5])
print("first :", sum(harvest))
print("second:", sum(harvest))    # empty, not an error
```

```
[1, 2, 3, 4, 5]
['A1', 'A2', 'B1']
first : 754.0
second: 0
```

- The generator `endless_numbers()` never ends, yet it is safe: it works only when asked. `itertools.islice(..., 5)` takes the first five and then stops asking.
- `yield from` passes along all items from another iterable, so `all_ponds()` joins two lists.
- **The trap:** the first `sum(harvest)` gives 754.0, but the second gives 0. A generator can be read only **once**. There is no error, just an empty result, which makes this bug slippery. If data must be read twice, build a new generator or save it to a list.

Chaining generators into a pipeline

The biggest strength of generators: several can be joined like water pipes. Each stage does one small job, and data flows one item at a time.

```
# Chapter 30 - Chaining generators into a pipeline
records = ["pond,A,540.5", "pond,B,broken", "pond,C,498.0", "", "pond,D,610.0"]

def non_empty(lines):
    for line in lines:
        if line.strip():
            yield line

def split_fields(lines):
    for line in lines:
        yield line.split(",")
```



```
def numbers_only(rows):
    for sector, unit, value in rows:
        try:
            yield unit, float(value)
        except ValueError:
            print("  skipped:", unit, value)

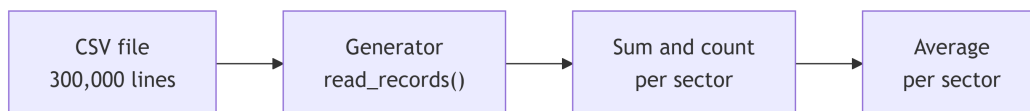
pipeline = numbers_only(split_fields(non_empty(records)))
print("pipeline built, nothing processed yet")
for unit, weight in pipeline:
    print(unit, weight)
```

```
pipeline built, nothing processed yet
A 540.5
  skipped: B broken
C 498.0
D 610.0
```

When `pipeline` is built, nothing has been processed yet (that is why “pipeline built, nothing processed yet” appears first). The work happens in the `for`. The empty line is filtered by `non_empty`, the line `pond,B,broken` is skipped by `numbers_only` (with the message “skipped”), and the rest flows to the end. There is no intermediate list.

Running Project I4: reading a big data file

The Series I4 project strengthens your field code, chapter by chapter. Step one: reading a large field-record file. The script below makes a fake file (sample numbers, fictional) for five sectors (forest, orchard, ricefield, pond, barn) with 300,000 lines, larger than 5 MB, then reads it with a generator.



Project flow: the big file is read one line at a time and summarised, without keeping everything.

```
# Chapter 30 - Running project I4: reading a big file with a generator
import random
import tracemalloc
from pathlib import Path

folder = Path(__file__).parent / "_tmp"
folder.mkdir(exist_ok=True)
file = folder / "big_field_data.csv"

# 1) Make a fake file (sample numbers, fictional)
ranges = {"forest": (20, 60), "orchard": (150, 350), "ricefield": (3000, 7000),
          "pond": (400, 700), "barn": (300, 450)}
rng = random.Random(42)
with open(file, "w", encoding="utf-8") as f:
    f.write("sector,unit,day,value\n")
    for i in range(300_000):
        sector = list(ranges)[i % 5]
        low, high = ranges[sector]
        f.write(f"{sector},{sector[0].upper()}{i % 7 + 1},{i // 5 + 1},{rng.uniform(low,
high):.1f}\n")
```

```

print("file is larger than 5 MB:", file.stat().st_size > 5_000_000)

# 2) A generator reads line by line
def read_records(path):
    with open(path, encoding="utf-8") as f:
        next(f) # skip the header
        for line in f:
            sector, unit, day, value = line.strip().split(",")
            yield sector, unit, int(day), float(value)

def average_per_sector(path):
    total, count = {}, {}
    for sector, _, _, value in read_records(path):
        total[sector] = total.get(sector, 0) + value
        count[sector] = count.get(sector, 0) + 1
    return {s: total[s] / count[s] for s in sorted(total)}, sum(count.values())

tracemalloc.start()
average, how_many = average_per_sector(file)
_, peak_gen = tracemalloc.get_traced_memory()
tracemalloc.stop()
print("records read:", how_many)
for sector, value in average.items():
    print(f"{sector:9} average {value:8.1f}")

# 3) For comparison: read everything into a list first
tracemalloc.start()
everything = list(read_records(file))
_, peak_list = tracemalloc.get_traced_memory()
tracemalloc.stop()
print("records in the list:", len(everything))
print("list approach peak memory is more than 20x the generator:", peak_list > 20
* peak_gen)

```

```

file is larger than 5 MB: True
records read: 300000
barn      average    375.2
forest    average     40.0
orchard    average    250.1
pond       average    550.2
ricefield  average   4996.2
records in the list: 300000
list approach peak memory is more than 20x the generator: True

```

Follow the result:

- The file is made with a fixed random seed (`random.Random(42)`), so the numbers are the same every run. It is placed in `_tmp/`, not in the script folder.
- Each sector gets 60,000 lines (300,000 divided by 5). The per-sector averages follow the ranges we set: ricefield around 5,000, pond around 550, and so on.
- The generator `read_records` uses `with open(...)` (Chapter 13): the file is closed automatically. The loop `for line in f` itself already reads the file **line by line**, without loading everything.
- For comparison: `list(read_records(file))` keeps all 300,000 records, and its peak memory is more than 20 times higher. Because only the comparison is printed, the output is stable.

TECHNICAL STUFF (feel free to skip)

`sys.getsizeof` measures only the object itself, **not** what it refers to. For a list, the size grows because a list really does hold a pointer to every item. For a generator, the size stays fixed because it keeps only its last state (the position in the function and its local variables). That is why this chapter uses `tracemalloc` to compare real memory use, and prints only the comparison.

Exercises

1. Write a generator function `evens_up_to(n)` that yields the even numbers from 2 to `n`. Print `list(evens_up_to(10))`.
2. Change the generator expression in `04_generator_expression.py` so it sums only the squares of odd numbers. Does the sum change? Why?
3. Add one stage to the pipeline in `06_generator_pipeline.py`: a generator `above(rows, limit)` that lets through only weights above 500. Attach it at the end of the pipeline.
4. **Challenge:** change `07_project_read_big.py` so it also finds the **largest** value per sector in a single pass, without keeping all the records.

Chapter Summary

- `for` uses the **iteration protocol**: `iter()` to ask for an iterator, `next()` for the next item, `StopIteration` when done.
- A **generator** is made with `yield`; the function **pauses** and resumes from its last position.
- A **generator expression** (`x for x in ...`) computes on demand, stays small, and saves memory.
- A generator can be read only **once**; an empty result on the second read is the classic trap.
- Several generators can be **chained into a pipeline** without intermediate lists.
- Project I4: the big file is read with a generator, and 300,000 records are summarised without piling up in memory.

Next, in Chapter 31: decorators and closures, a way to “wrap” a function with extras (logging, retrying) without changing what is inside.

Chapter 31 — Decorators and Closures: Wrapping Functions with Extras

Series I4 — Advanced Python Language

In this chapter, you will:

- Understand that functions are **objects** that can be stored and passed around
- Build a **closure**: a function that remembers the variables around it
- Write **decorators**, including ones with arguments and `functools.wraps`
- Recognise a common trap and a ready-made decorator (`functools.cache`)
- Add logging and retrying to project functions without changing what is inside them

All scripts for this chapter are in the `code/ch31/` folder. On the web version, you can also [download them all at once \(zip\)](#).

In Chapter 30, your project learned to read big files. Now imagine you have ten functions that fetch data from various places, and you want: (1) a log entry each time a function starts and finishes, (2) automatic repeating when the network drops. Copying the logging and repeating code into ten functions is boring and error-prone. Better to “wrap” each function in an extra layer, like wrapping a gift: the contents stay the same, but there is a layer on the outside. That is a **decorator**.

A decorator = wrapping layers around a function



The call passes through the layers from outside in, the result comes back from inside out

Figure 0.1: A decorator = wrapping layers around a function: the call passes through the layers, the result comes back out.

Functions are objects

The foundation of decorators: in Python, a function is just one kind of object, like a number or a piece of text. It can be given another name, stored in a dict, and passed to another function.

```
# Chapter 31 - Functions are objects
def biomass_kg(count, weight_g):
    return count * weight_g / 1000

tool = biomass_kg          # copies the "name", does not call it
print(tool(800, 540.5))
```

```

print(tool is biomass_kg)
print(type(biomass_kg).__name__)

# Functions can be stored in a dict and picked by name
formulas = {
    "biomass": biomass_kg,
    "area": lambda length, width: length * width,
}
print(formulas["area"](20, 15))

# Functions can be passed to other functions
def apply(function, *arguments):
    return function(*arguments)

print(apply(biomass_kg, 1000, 500))

```

```

432.4
True
function
300
500.0

```

- `tool = biomass_kg` (without brackets) **does not call** the function; it merely adds another name for the same object. Proof: `tool is biomass_kg` is `True`.
- The object's type is `function`.
- In the dict `formulas`, a function is picked by name: `formulas["area"](20, 15)` gives 300.
- `apply(biomass_kg, 1000, 500)` passes a function as an ingredient. `*arguments` gathers the remaining arguments, and `function(*arguments)` unpacks them again. The result is 500.0.

Closures: a function that remembers

A function may be made **inside** another function. The inner function can use variables owned by the outer function, even after the outer function has finished. The combination “function + the variables it remembers” is called a **closure**.

```

# Chapter 31 - Closure: a function that remembers
def make_counter(start=0):
    total = start

    def add(amount):
        nonlocal total          # without this, total would be a new variable
        total += amount
        return total

    return add

orchard_harvest = make_counter()
pond_harvest = make_counter(100)
print(orchard_harvest(10))
print(orchard_harvest(5))
print(pond_harvest(1))          # another counter, separate memory

content = orchard_harvest.__closure__[0].cell_contents
print("memory of orchard_harvest:", content)

```

```
10
15
101
memory of orchard_harvest: 15
```

- `make_counter()` returns the function `add`. That function “carries” the variable `total`.
- `orchard_harvest` and `pond_harvest` have **separate** memories: 10, then 15 for the orchard, while the pond starts from 100 and becomes 101.
- **nonlocal** `total` is required when the inner function wants to **change** the outer variable. Without it, `total += amount` is treated as making a new local variable and raises an error (remember the scope discussion in Chapter 10).
- `__closure__[0].cell_contents` shows that “memory” directly: 15.

Your first decorator

A decorator is a function that **takes a function** and **returns a new function** (the wrapper). The `@` sign is only shorthand.

```
# Chapter 31 - A first decorator (not perfect yet)
def log_call(function):
    def wrapper(*args, **kwargs):
        print(f"calling {function.__name__}")
        result = function(*args, **kwargs)
        print(f"done, result = {result}")
        return result
    return wrapper

@log_call
def biomass_kg(count, weight_g):
    """Pond biomass in kg."""
    return count * weight_g / 1000

biomass_kg(800, 540.5)

# @log_call is the same as: biomass_kg = log_call(biomass_kg)
print(biomass_kg.__name__)      # the real identity is lost
print(biomass_kg.__doc__)
```

```
calling biomass_kg
done, result = 432.4
wrapper
None
```

- `@log_call` above `def biomass_kg` means the same as `biomass_kg = log_call(biomass_kg)`.
- Every call now goes through `wrapper`: prints “calling”, runs the real function, prints the result, then returns it.
- `*args, **kwargs` lets the wrapper accept **any arguments** and pass them along unchanged.
- There is a flaw: `biomass_kg.__name__` has become `wrapper` and its docstring is gone (`None`). The function lost its identity.

functools.wraps: keeping the identity

The cure is one line: `@wraps(function)` on the wrapper.

```
# Chapter 31 - functools.wraps keeps the function's identity
from functools import wraps
```

```

def log_call(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        print(f"calling {function.__name__}")
        return function(*args, **kwargs)
    return wrapper

@log_call
def biomass_kg(count, weight_g):
    """Pond biomass in kg."""
    return count * weight_g / 1000

print(biomass_kg(800, 540.5))
print(biomass_kg.__name__)
print(biomass_kg.__doc__)
print(biomass_kg.__wrapped__ is not None)

```

```

calling biomass_kg
432.4
biomass_kg
Pond biomass in kg.
True

```

Now `__name__` and `__doc__` are back to the original, and `__wrapped__` keeps a reference to the function before wrapping. Make it a habit: **every decorator always uses `@wraps`**. This matters for error messages, documentation, and tools such as `pytest`.

A decorator that takes arguments

When a decorator needs settings (for example “repeat 3 times”), one more layer is needed: the outer function takes the settings and returns the real decorator.

```

# Chapter 31 - A decorator that takes arguments
from functools import wraps

def repeat(times):
    """Decorator factory: repeats the call `times` times."""
    def decorator(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            result = None
            for i in range(times):
                result = function(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print("Hello,", name)

greet("Farmer")

```

```
# Stacking decorators: the one closest to def is applied first
def bold(f):
    return lambda: "<b>" + f() + "</b>"

def italic(f):
    return lambda: "<i>" + f() + "</i>"

@bold
@italic
def title():
    return "Harvest"

print(title())
```

```
Hello, Farmer
Hello, Farmer
Hello, Farmer
<b><i>Harvest</i></b>
```

Three levels: `repeat(3)` → returns `decorator` → which takes `greet` and returns `wrapper`. `@repeat(3)` runs the first level right away, and the result is applied as the decorator.

The bottom part shows **stacked decorators**: the one closest to `def` is applied first. `@italic` wraps `title` first, then `@bold` wraps that result, so the `<b>` tag ends up outside and `<i>` inside: `<b><i>Harvest</i></b>`.

A ready-made decorator and a trap

The standard library offers many decorators. One of the most useful is `functools.cache`, which remembers results so a call with the same arguments is not computed again.

```
# Chapter 31 - A ready-made decorator: functools.cache
from functools import cache

calls = 0

@cache
def feed_price_per_kg(code):
    global calls
    calls += 1          # counts how often the body really runs
    return {"P1": 8500, "P2": 9200}[code]

print(feed_price_per_kg("P1"))
print(feed_price_per_kg("P1"))  # taken from memory
print(feed_price_per_kg("P2"))
print("function body ran:", calls, "times")
```

```
8500
8500
9200
function body ran: 2 times
```

Called three times, but the function body ran only **2 times**: the second call for `"P1"` came from memory. This suits slow functions whose result is always the same for the same input. Do not use it for functions whose result changes (for example reading today's price).

Now the most common trap: **forgetting the brackets** on a decorator with arguments.


```
# Chapter 31 - Trap: a decorator with arguments used without brackets
```

```
def repeat(times):  
    def decorator(function):  
        def wrapper(*args, **kwargs):  
            for _ in range(times):  
                function(*args, **kwargs)  
            return wrapper  
        return decorator
```

```
@repeat          # should be @repeat(3)  
def greet():  
    print("Hello")
```

```
greet()
```

```
Traceback (most recent call last):  
  File "07_forgot_brackets.py", line 16, in <module>  
    greet()  
    ~~~~~^^
```

```
TypeError: repeat.<locals>.decorator() missing 1 required positional argument:  
'function'
```

@repeat without (3) means `greet = repeat(greet)`. The argument `times` now holds the function `greet`, and what comes back is `decorator`, not a wrapper. When `greet()` is called, it is really `decorator()` that runs, and it lacks the argument `function`. The error message (`repeat.<locals>.decorator() missing 1 required positional argument: 'function'`) is the telltale sign.

⚠ WARNING

A decorator with arguments is **always** written with brackets, even if the arguments are empty or default. If the error reads `<locals>.decorator() missing ... 'function'`, you almost certainly forgot the brackets.

Running Project I4: logging and retrying

Now combine them: two decorators, `log_call` (logging) and `retry` (repeating on `ConnectionError`), applied to a rainfall-fetching function. A fake network: it fails twice, then succeeds. The log uses the `logging` module (Chapter 19) with a format without a clock, so the output is stable. The pause between attempts is deliberately tiny (0.01 seconds).

```
# Chapter 31 - Running project I4: decorators for logging and retrying
```

```
import logging  
import sys  
import time  
from functools import wraps  
  
logging.basicConfig(stream=sys.stdout, level=logging.INFO, format="%(levelname)s  
%(message)s")  
log = logging.getLogger("field")  
  
def log_call(function):  
    @wraps(function)  
    def wrapper(*args, **kwargs):  
        log.info("start %s", function.__name__)  
        result = function(*args, **kwargs)  
        log.info("finish %s", function.__name__)
```

```

        return result
    return wrapper

def retry(maximum=3, pause=0.01, errors=(ConnectionError,)):
    def decorator(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            for attempt in range(1, maximum + 1):
                try:
                    return function(*args, **kwargs)
                except errors as e:
                    log.warning("attempt %d of %d failed: %s", attempt, maximum, e)
                    if attempt == maximum:
                        raise
                    time.sleep(pause)
            return wrapper
        return decorator

outage = {"left": 2}  # fake network: fails twice, then works

@log_call
@retry(maximum=3)
def fetch_rainfall(station):
    if outage["left"] > 0:
        outage["left"] -= 1
        raise ConnectionError("connection dropped")
    return {"station": station, "mm": 12.5}

print(fetch_rainfall("S1"))

outage["left"] = 5  # now the outage outlasts the retries
try:
    fetch_rainfall("S2")
except ConnectionError as e:
    print("giving up:", e)

```

```

INFO start fetch_rainfall
WARNING attempt 1 of 3 failed: connection dropped
WARNING attempt 2 of 3 failed: connection dropped
INFO finish fetch_rainfall
{'station': 'S1', 'mm': 12.5}
INFO start fetch_rainfall
WARNING attempt 1 of 3 failed: connection dropped
WARNING attempt 2 of 3 failed: connection dropped
WARNING attempt 3 of 3 failed: connection dropped
giving up: connection dropped

```

Follow along:

- First call: start is logged, attempts 1 and 2 fail (WARNING), attempt 3 succeeds, then finish is logged. The result is {'station': 'S1', 'mm': 12.5}.
- The decorator order matters: @log_call outside, @retry inside. So the start and finish logs appear only **once** per call, even though three attempts happen inside.
- Second call: the outage is longer than three attempts, so the error is raised again (raise). Notice: there is no finish, because the function did not end well. The program catches it outside with try/except and prints "giving up".

💡 TIP

Retrying suits only **temporary** errors (dropped network, busy server). Do not repeat errors that are certain to repeat (bad data): it just wastes time. That is why `retry` catches only the error types you name.

Exercises

1. Write a closure `make_multiplier(factor)` that returns a function multiplying a number by `factor`. Make `times_three` and `times_ten`, then try both.
2. Write a decorator `count_calls` that stores the number of calls in a function attribute (`wrapper.count`). Apply it to one function, call it four times, and print `count`.
3. Change `05_decorator_arguments.py`: add `@repeat(2)` to another function that prints a pond name, and make sure its `__name__` stays correct.
4. **Challenge:** write a decorator `at_least(limit)` that raises `ValueError` if the function's result is below `limit`. Apply it to the `biomass` function, and test with one input that passes and one that fails.

Chapter Summary

- A function is an **object**: it can get another name, be stored, and be passed to other functions.
- A **closure** = an inner function that remembers the outer function's variables; change them with `nonlocal`.
- A **decorator** = a function that takes a function and returns its wrapper; `@name` is shorthand.
- Always use `@wraps` so the original function's name and docstring survive.
- A decorator **with arguments** needs three levels and **must have brackets** when applied.
- Project I4: the data-fetching function now has logging and retrying without changing what is inside it.

Next, in Chapter 32: context managers, a way to make sure resources (files, connections) are always closed neatly.

Chapter 32 — Context Managers: Open, Use, and Surely Close

Series I4 — Advanced Python Language

In this chapter, you will:

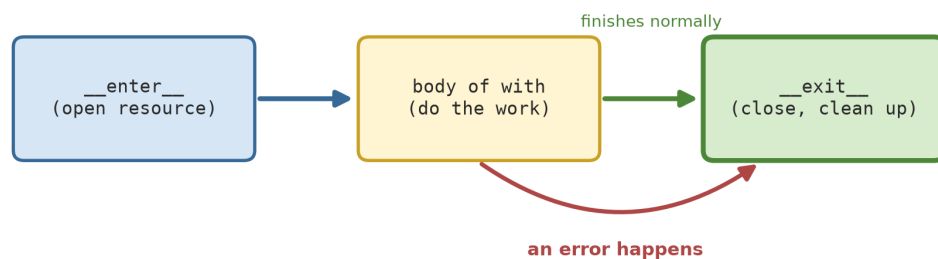
- Understand why `with` is safer than `open()` followed by `close()`
- Build a context manager with a class (`__enter__` and `__exit__`)
- Build one more briefly with `@contextmanager`
- Use `contextlib` tools (`suppress`, `redirect_stdout`, `ExitStack`)
- Write a report safely: the old file is not damaged if writing fails

All scripts for this chapter are in the `code/ch32/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Have you ever forgotten to switch off the pond pump after filling it with water? The pump keeps running, electricity is wasted, or the pond overflows. Programs have a similar problem: a file opened and never closed, a connection left hanging. Worse still, if an error strikes in the middle of the work, the `close()` line at the end is never reached.

Python has a solution: the `with` statement. It promises that whatever happens inside the block, the “close and clean up” job **will** be done.

with: the door is opened and GUARANTEED to be closed



The cleanup in `__exit__` runs on success and on failure alike

Figure 0.1: `with`: the door is opened, the work is done, then the door is made sure to close, whether it succeeded or failed.

with on files

You have used `with open(...)` since Chapter 13. Compare it with the old way:

```
# Chapter 32 - with closes the file automatically
from pathlib import Path

folder = Path(__file__).parent / "_tmp"
folder.mkdir(exist_ok=True)
path = folder / "notes.txt"

# The old way: you must remember close()
```

```
f = open(path, "w", encoding="utf-8")
f.write("orchard A harvest: 245.5 kg\n")
print("before close, closed?", f.closed)
f.close()
print("after close, closed?", f.closed)

# The with way: closes itself, even when an error happens
with open(path, encoding="utf-8") as f:
    content = f.read()
    print("inside with, closed?", f.closed)
print("outside with, closed?", f.closed)
print(content.strip())
```

```
before close, closed? False
after close, closed? True
inside with, closed? False
outside with, closed? True
orchard A harvest: 245.5 kg
```

- The old way: `f.closed` is `False` until you yourself call `f.close()`. If an error occurs before `close()`, the file stays open.
- The `with` way: inside the block, the file is open (`False`). As soon as the block ends, it is closed (`True`), without you writing `close()`.
- Note that the variable `f` still exists after the block, but the file is already closed.

Behind the scenes: `__enter__` and `__exit__`

`with` works on any object that has two special methods: `__enter__` (run when entering the block; its value goes into the variable after `as`) and `__exit__` (run when leaving the block, **always**, even when an error occurs). Such an object is called a **context manager**.

```
# Chapter 32 - A context manager as a class: __enter__ and __exit__
class Door:
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        print(f"open door {self.name}")
        return self                # the value after the word "as"

    def __exit__(self, error_type, error, trace):
        print(f"close door {self.name}")
        return False               # False: the error (if any) keeps travelling

with Door("barn") as d:
    print("working inside", d.name)

print("--- now with an error ---")
try:
    with Door("pond"):
        print("working...")
        raise ValueError("pump is off")
except ValueError as e:
    print("error caught outside:", e)
```

```
open door barn
working inside barn
close door barn
```

```

--- now with an error ---
open door pond
working...
close door pond
error caught outside: pump is off

```

- In the first attempt the order is: open, work, close.
- In the second attempt, the error `ValueError("pump is off")` happens inside the block. The door is **still closed** ("close door pond" is printed) before the error shows up outside and is caught by `except`.
- `__exit__` receives three pieces of error information (type, value, trace; they are `None` if there is no error). Returning `False` means "let the error travel on".

The short way: `@contextmanager`

Writing a class with two methods is long. `contextlib.contextmanager` turns a **generator function** (Chapter 30) into a context manager: the code before `yield` becomes `__enter__`, the code after it becomes `__exit__`.

```

# Chapter 32 - @contextmanager: a context manager from a generator function
from contextlib import contextmanager

@contextmanager
def work_session(name):
    print(f"[{name}] start")
    try:
        yield name.upper()           # the body of the with block runs here
    finally:
        print(f"[{name}] finished and cleaned up")

with work_session("fertilising") as code:
    print("working with code", code)

try:
    with work_session("harvesting"):
        print("working...")
        1 / 0
except ZeroDivisionError:
    print("error caught, but the cleanup still ran")

```

```

[fertilising] start
working with code FERTILISING
[fertilising] finished and cleaned up
[harvesting] start
working...
[harvesting] finished and cleaned up
error caught, but the cleanup still ran

```

- `yield name.upper()` hands control to the `with` block and gives the value `FERTILISING` to the variable `code`.
- The `try ... finally` around `yield` guarantees the cleanup. In the second attempt `1 / 0` happens, yet "finished and cleaned up" is still printed, and then the error is caught outside.

TIP

Always wrap the `yield` in `@contextmanager` with `try ... finally` if there is cleanup. Without `finally`, if the `with` block raises an error, the cleanup lines after `yield` **do not run**.

`__exit__` may swallow an error

If `__exit__` returns `True`, the error counts as **already handled** and the program carries on as if nothing happened. This is powerful but dangerous.

```
# Chapter 32 - __exit__ may swallow an error (use with care)
from contextlib import suppress

class IgnoreZero:
    def __enter__(self):
        return self

    def __exit__(self, error_type, error, trace):
        if error_type is ZeroDivisionError:
            print("division by zero ignored")
            return True          # True: the error counts as handled
        return False

with IgnoreZero():
    print(10 / 0)
print("program continues")

# The standard library already has a similar tool
with suppress(FileNotFoundError):
    open("does_not_exist.txt", encoding="utf-8")
print("continues again")
```

```
division by zero ignored
program continues
continues again
```

In the first block, `10 / 0` raises `ZeroDivisionError`, but `__exit__` returns `True`, so “program continues” still appears. The standard library offers a tidier version: `contextlib.suppress(FileNotFoundError)`, which swallows **only** the error type you name.

WARNING

Swallowing errors can hide bugs. Swallow only error types you **expect** and that are truly harmless (for example a file that is allowed not to exist). Never swallow all errors.

Ready-made tools in `contextlib`

```
# Chapter 32 - Ready-made tools in contextlib and other modules
import io
from contextlib import ExitStack, redirect_stdout
from pathlib import Path

# redirect_stdout: capture what gets printed
capture = io.StringIO()
with redirect_stdout(capture):
    print("this does not show on screen")
print("captured:", repr(capture.getvalue()))

# ExitStack: the number of files is only known while running
folder = Path(__file__).parent / "_tmp"
folder.mkdir(exist_ok=True)
names = ["pond_a.txt", "pond_b.txt", "pond_c.txt"]
```

```
with ExitStack() as stack:
    files = [stack.enter_context(open(folder / n, "w", encoding="utf-8")) for n in names]
    for i, f in enumerate(files):
        f.write(f"data of pond number {i + 1}\n")
        print("open at the same time:", len(files))
    print("all closed:", all(f.closed for f in files))
```

```
captured: 'this does not show on screen\n'
open at the same time: 3
all closed: True
```

- `redirect_stdout` redirects what `print` writes into an `io.StringIO` container. The screen shows nothing, but the content can be read afterwards ('this does not show on screen\n'). Useful for testing functions that print.
- `ExitStack` is used when the number of resources is only known while the program runs. Here three files are opened through `enter_context`, and all three are closed together when the block ends.

The trap: `with` on an object that is not a context manager

```
# Chapter 32 - Wrong: with on an object that is not a context manager
class Warehouse:
    pass

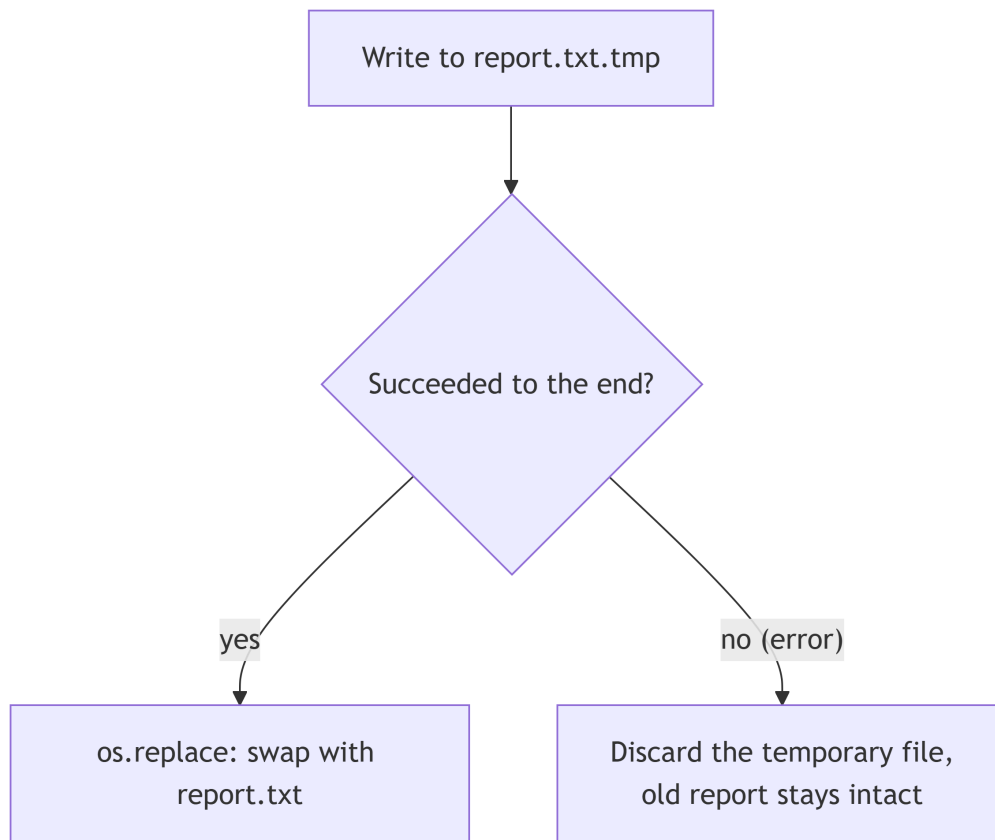
with Warehouse() as w:
    print("we will never get here")
```

```
Traceback (most recent call last):
  File "06_not_a_context_manager.py", line 6, in <module>
    with Warehouse() as w:
        ~~~~~^
TypeError: 'Warehouse' object does not support the context manager protocol (missed
__exit__ method)
```

The error message is clear: the class `Warehouse` has no `__exit__` (and no `__enter__`), so it does not support the “context manager protocol”. The fix: add those two methods, or use `@contextmanager`.

Running Project I4: writing a report safely

A real scenario: you write a daily report to a file. In the middle of writing, the power goes out or the program errors. A half-written file overwrites the old, complete report. The common solution: write to a **temporary file**, and only if the whole write succeeds, **swap** it with the real file.



Safe writing: the temporary file replaces the real report only if everything succeeded.

```
# Chapter 32 - Running project I4: writing a report safely
import os
from contextlib import contextmanager
from pathlib import Path

folder = Path(__file__).parent / "_tmp"
folder.mkdir(exist_ok=True)

@contextmanager
def safe_write(target):
    """Writes to a temporary file; only on success does it replace the real one."""
    temporary = Path(str(target) + ".tmp")
    f = open(temporary, "w", encoding="utf-8")
    try:
        yield f
    except BaseException:
        f.close()
        temporary.unlink()  # failed: throw away the half-finished file
        print("failed, temporary file discarded")
        raise
    else:
        f.close()
        os.replace(temporary, target)  # success: swap in one step
        print("success, report saved")
```

```

report = folder / "report.txt"

with safe_write(report) as f:
    f.write("Orchard harvest: 1030.0 kg\n")

try:
    with safe_write(report) as f:
        f.write("Half-finished data...\n")
        raise RuntimeError("power cut")
except RuntimeError as e:
    print("error:", e)

print("report content still intact:", report.read_text(encoding="utf-8").strip())
print("temporary file left over:", Path(str(report) + ".tmp").exists())

```

```

success, report saved
failed, temporary file discarded
error: power cut
report content still intact: Orchard harvest: 1030.0 kg
temporary file left over: False

```

Follow along:

- The first write succeeds: “success, report saved”, and the content `Orchard harvest: 1030.0 kg` replaces the file.
- The second write simulates a power cut (`RuntimeError`) after writing half the content. In the `except BaseException` block, the temporary file is closed and **discarded**, then the error is raised again (`raise`) so it is caught outside.
- Final result: the report content **stays intact** as before the second write, and no `.tmp` file is left behind (`False`).
- `os.replace` replaces the target file. According to the Python documentation, if successful this operation is **atomic** (it does not get split in the middle), but it can fail if the source and target are on different filesystems. That is why the temporary file is placed in the same folder as the target.

TECHNICAL STUFF (feel free to skip)

`BaseException` is broader than `Exception`: it also covers `KeyboardInterrupt` (you press Ctrl+C) and `SystemExit`. For cleanup, we want to be caught in all those cases, and then **raise again** so the error still reaches the caller. To handle ordinary errors, `Exception` is enough.

Exercises

1. Write a context manager class `Recorder` that prints “start” in `__enter__` and “finish” in `__exit__`. Use it with `with` and add one `print` inside.
2. Change `03_contextmanager.py`: add a third `work_session("watering")` that suffers a `ValueError`, and catch that error outside. Is “cleaned up” still printed?
3. Use `contextlib.suppress(FileNotFoundError)` to delete the file `_tmp/does_not_exist.txt` with `Path.unlink()` without an error if the file really is absent.
4. **Challenge:** write a context manager `log_step(name)` using `@contextmanager` that prints “[name] start” and “[name] finish”; if an error occurs, print “[name] FAILED” and then raise the error again. Test it with one successful block and one failing block.

Chapter Summary

- **with** guarantees cleanup (closing files, releasing resources) even when an error occurs.
- A context manager = an object with `__enter__` and `__exit__`; `__exit__` always runs.

- `@contextmanager` turns a generator function into a context manager; wrap the `yield` in `try ... finally`.
- An `__exit__` that returns `True` **swallows** the error; use `suppress` only for expected error types.
- `contextlib` provides `suppress`, `redirect_stdout`, and `ExitStack` (a dynamic number of resources).
- Project I4: the report is written through a temporary file, so the old report stays intact if writing fails.

Next, in Chapter 33: type hints, a way to tell people and checking tools what kind of data is expected.

Chapter 33 — Type Hints: Labelling Data, Checking Before Running

Series I4 — Advanced Python Language

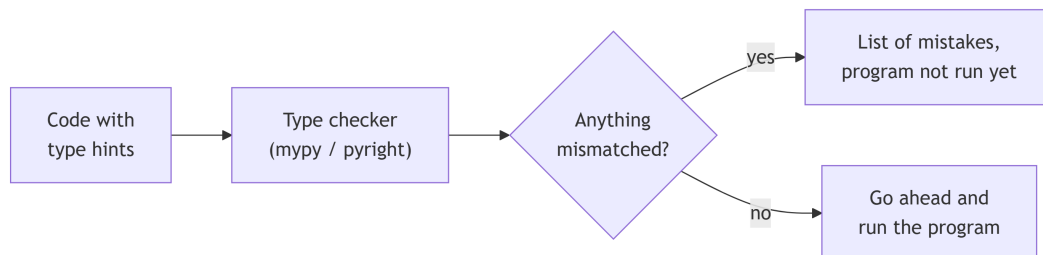
In this chapter, you will:

- Write **type annotations** (*type hints*) on variables and functions
- Use types for collections, choices (`X | None`), functions, and generics
- Meet `TypedDict`, `Literal`, and `Protocol`
- Run the real **mypy** and read its output
- Understand a Python 3.14 change: annotations are evaluated later

All scripts for this chapter are in the `code/ch33/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Imagine a feed store with unlabelled sacks. You must open each sack to know what is inside, and sometimes you grab the wrong one. Now imagine every sack labelled “fish pellets, 25 kg”. You know at once, and an inspector can flag a misplaced sack before the feed is used.

A **type hint** is that label on code: a note about the kind of data a variable or function expects. Python itself **does not enforce it** while the program runs, but people, editors, and checking tools can use it to find mistakes **before** the program runs.



Type hints are used by checking tools (mypy, pyright) to find mistakes before the program runs.

Basic annotations

```
# Chapter 33 - Type annotations: notes for humans and tools, not enforcement
def biomass_kg(count: int, weight_g: float) -> float:
    return count * weight_g / 1000

age_days: int = 120
pond_name: str = "A"

print(biomass_kg(800, 540.5))
print(biomass_kg.__annotations__)

def double(count: int) -> int:
    return count * 2
```

```
# Python does NOT check annotations while the program runs
print(double(21))
print(double("ab"))           # not an int, yet it still runs
```

```
432.4
{'count': <class 'int'>, 'weight_g': <class 'float'>, 'return': <class 'float'>}
42
abab
```

- `count: int` means “count is expected to be a whole number”. -> `float` at the end means “the function returns a decimal number”.
- Variables can be labelled too: `age_days: int = 120`.
- Annotations are stored in `__annotations__` as notes.
- **Python does not check them while running.** `double("ab")` is not an `int`, yet it runs and gives `abab` (text times two repeats the text). No error. The checking is done by a separate tool, not by Python.

i REMEMBER

A type hint is a **note**, not a **doorkeeper**. Without a checking tool, a wrong type is only discovered while the program runs, or not discovered at all like the `abab` above. For a doorkeeper at run time, see Pydantic in Chapter 34.

Types for collections, choices, and functions

```
# Chapter 33 - Types for collections, choices, and functions
from collections.abc import Callable, Iterator

def average(numbers: list[float]) -> float:
    return sum(numbers) / len(numbers)

def find_pond(data: dict[str, float], name: str) -> float | None:
    """Returns the pond weight, or None if it does not exist."""
    return data.get(name)

def point(x: float, y: float) -> tuple[float, float]:
    return (x, y)

def apply(function: Callable[[float], float], values: list[float]) -> list[float]:
    return [function(v) for v in values]

def countdown(n: int) -> Iterator[int]:
    while n > 0:
        yield n
        n -= 1

weights = {"A": 540.5, "B": 498.0}
print(average([540.5, 498.0, 610.0]))
print(find_pond(weights, "A"), find_pond(weights, "Z"))
print(point(1.5, 2.0))
```

```
print(apply(lambda g: g / 1000, [540.5, 498.0]))
print(list(countdown(3)))
```

```
549.5
540.5 None
(1.5, 2.0)
[0.5405, 0.498]
[3, 2, 1]
```

Written as	Meaning
<code>list[float]</code>	a list whose items are decimals
<code>dict[str, float]</code>	a dict with text keys and decimal values
<code>tuple[float, float]</code>	a tuple of two decimals
<code>float None</code>	a decimal or <code>None</code> (meaning “may be empty”)
<code>Callable[[float], float]</code>	a function taking one decimal and returning a decimal
<code>Iterator[int]</code>	something you can <code>next()</code> that gives whole numbers

`find_pond(weights, "Z")` returns `None` because pond Z does not exist; that is why its type is `float | None`. Writing `float | None` uses the `|` sign (Python 3.10 and up). `Callable` and `Iterator` come from `collections.abc`. Note: `countdown` is a generator (Chapter 30), and the fitting type for a simple generator is `Iterator[int]`.

Generics: types that follow the content

The function `first(items)` can be used on a list of decimals or a list of text. The result type must follow: decimal if given a list of decimals, text if given a list of text. That is the job of **generics**, with a type variable `T`.

```
# Chapter 33 - Generics: types that follow the content
from collections.abc import Sequence

def first[T](items: Sequence[T]) -> T:
    """Returns the first element; its type matches the list content."""
    return items[0]

print(first([540.5, 498.0]))      # T = float
print(first(["A", "B"]))         # T = str

class Warehouse[T]:
    """A place to store items of the same type."""

    def __init__(self) -> None:
        self.content: list[T] = []

    def store(self, item: T) -> None:
        self.content.append(item)

    def take(self) -> T:
        return self.content.pop()
```

```
weight_store = Warehouse[float]()
weight_store.store(540.5)
print(weight_store.take())
```

```
540.5
A
540.5
```

The syntax `def first[T](...)` and `class Warehouse[T]` is the way of writing introduced in Python 3.12 (before that, `TypeVar` and `Generic` were used). `Warehouse[float]()` makes a decimals-only warehouse. A checking tool will then complain if you store text in it.

TypedDict, Literal, and Protocol

Field data often takes the form of a dict. `TypedDict` states which keys exist and their types. `Literal` limits the choices to certain values. `Protocol` states “anything that has this” without requiring inheritance (called *duck typing*: “if it walks like a duck and quacks like a duck, it is a duck”).

```
# Chapter 33 - TypedDict, Literal, and Protocol
from dataclasses import dataclass
from typing import Literal, Protocol, TypedDict

Sector = Literal["forest", "orchard", "ricefield", "pond", "barn"]

class Record(TypedDict):
    sector: Sector
    unit: str
    value: float

@dataclass
class Plot:
    name: str
    area_ha: float

class HasName(Protocol):
    """A 'duck' type: anything that has a name attribute."""
    name: str

def greet(thing: HasName) -> str:
    return "Hello, " + thing.name

r: Record = {"sector": "pond", "unit": "A", "value": 540.5}
print(r["sector"], r["value"])
print(greet(Plot("Block A", 1.5)))
```

```
pond 540.5
Hello, Block A
```

`Record` is a dict with three keys, and `sector` may only be one of five values. `greet()` accepts anything with a `name` attribute: here `Plot`, a dataclass (Chapter 16), inheriting from nothing. While running, everything is an ordinary dict and an ordinary object; the labels are only for the checker.

Running the real mypy

mypy is a type-checking tool. The script below writes a small file that is **deliberately wrong**, runs mypy on it, and shows the real output.

```
# Chapter 33 - Running the real mypy on a small file
import subprocess
import sys
from pathlib import Path

folder = Path(__file__).parent / "_tmp"
folder.mkdir(exist_ok=True)
example = folder / "wrong_example.py"
example.write_text('''def biomass_kg(count: int, weight_g: float) -> float:
    return count * weight_g / 1000

total: int = biomass_kg(800, 540.5)
print(biomass_kg("800", 540.5))
print(biomass_kg(800))
name: str = None
''', encoding="utf-8")

result = subprocess.run(
    [sys.executable, "-m", "mypy", "--no-color-output", "--no-error-summary",
     "--cache-dir", str(folder / ".mypy_cache"), "wrong_example.py"],
    cwd=folder, capture_output=True, text=True,
)
print(result.stdout.strip())
print("mypy exit code:", result.returncode)
```

```
wrong_example.py:5: error: Incompatible types in assignment (expression has type
"float", variable has type "int") [assignment]
wrong_example.py:6: error: Argument 1 to "biomass_kg" has incompatible type "str";
expected "int" [arg-type]
wrong_example.py:7: error: Missing positional argument "weight_g" in call to
"biomass_kg" [call-arg]
wrong_example.py:8: error: Incompatible types in assignment (expression has type
"None", variable has type "str") [assignment]
mypy exit code: 1
```

Mypy finds four mistakes **without running** the file:

- Line 5: `biomass_kg` returns a `float`, but it is stored in a variable promised to be `int`.
- Line 6: the first argument is the text `"800"`, while an `int` is expected.
- Line 7: the argument `weight_g` is missing.
- Line 8: `None` is given to a variable of type `str`.

The code in square brackets at the end (`[assignment]`, `[arg-type]`, `[call-arg]`) is the name of the error category. The mypy exit code 1 means “there are problems”, which is useful when mypy is run automatically (for example in a pre-commit check, Chapters 19 and 20).

TIP

Start small. Add type hints to the most used functions, run mypy, fix, then expand. mypy’s `--strict` option is stricter (it demands types on almost every function); it suits a project that is already tidy.

Pyright, another checking tool

Besides mypy, there is **pyright**, a type checker made by Microsoft. On its project page, pyright is described as a static type checker for Python offered as a command-line tool and a Visual Studio Code extension.

Pyright is **not run in this book**; how to install it and how its behaviour differs from mypy have not been verified on the author's computer, so please consult its own documentation if you want to try it. The concepts you learn here (annotations, generics, TypedDict) apply equally to both.

Python 3.14: annotations are evaluated later

In the past, an annotation such as `x: SomeClass` was evaluated immediately when `def` ran. If `SomeClass` was not yet defined, an error occurred, so people wrote `"SomeClass"` (in quotes) or `from __future__ import annotations`. Since **Python 3.14**, annotations are stored and evaluated **only when needed** (source: the “What’s New in Python 3.14” documentation, section on deferred annotations). As a result, a reference to a name that does not exist yet no longer needs quotes. The new module `annotationlib` reads annotations in three formats.

```
# Chapter 33 - Python 3.14: annotations are evaluated later
from annotationlib import Format, get_annotations

def compute(x: NotCreatedYet) -> None:    # this name does NOT exist yet
    pass

print("function was created without any error")
print(get_annotations(compute, format=Format.STRING))
reference = get_annotations(compute, format=Format.FORWARDREF)["x"]
print(type(reference).__name__, reference.__forward_arg__)
try:
    get_annotations(compute, format=Format.VALUE)
except NameError as e:
    print("NameError:", e)

class NotCreatedYet:
    pass

print(get_annotations(compute, format=Format.VALUE))    # now the name exists
```

```
function was created without any error
{'x': 'NotCreatedYet', 'return': 'None'}
ForwardRef NotCreatedYet
NameError: name 'NotCreatedYet' is not defined
{'x': <class '__main__.NotCreatedYet'>, 'return': None}
```

- `compute(x: NotCreatedYet)` is created successfully even though `NotCreatedYet` does not exist. On older Pythons this is an immediate `NameError`.
- The `STRING` format shows the annotation as text, `FORWARDREF` as a reference (`ForwardRef`) for a name that does not exist yet, and `VALUE` really evaluates the value.
- The `VALUE` format before the class exists gives a `NameError`. After `NotCreatedYet` is defined, `VALUE` succeeds and returns the class.

TECHNICAL STUFF (feel free to skip)

This script runs only on Python 3.14 or later, because `annotationlib` is new there and the behaviour “the function is created even though the name is missing” is also specific to 3.14. Readers still on an older Python should keep writing forward references in quotes. The code `from __future__ import annotations` is still recognised and keeps the old behaviour.

Running Project I4: typing the project code

We add type hints to the record-reading functions from Chapter 30, then check with mypy in strict mode. The script writes a small module `field.py` into `_tmp/`, runs mypy, **inserts one deliberate mistake**, runs mypy again, then runs the module.

```
# Chapter 33 - Running project I4: typing the project code, checked by mypy
import subprocess
import sys
from pathlib import Path

folder = Path(__file__).parent / "_tmp"
folder.mkdir(exist_ok=True)
module = folder / "field.py"
module.write_text('''from collections.abc import Iterable, Iterator
from typing import Literal, TypedDict, cast

Sector = Literal["forest", "orchard", "ricefield", "pond", "barn"]

class Record(TypedDict):
    sector: Sector
    unit: str
    value: float

def read(lines: Iterable[str]) -> Iterator[Record]:
    for line in lines:
        sector, unit, value = line.split(",")
        yield {"sector": cast(Sector, sector), "unit": unit, "value": float(value)}

def average(records: Iterable[Record]) -> float | None:
    values = [r["value"] for r in records]
    return sum(values) / len(values) if values else None

print(average(read(["pond,A,540.5", "pond,B,498.0"])))
print(average(read([])))
''', encoding="utf-8")

def mypy(*options: str) -> None:
    result = subprocess.run(
        [sys.executable, "-m", "mypy", "--no-color-output", "--cache-dir", str(folder /
        ".mypy_cache"),
        *options, "field.py"],
        cwd=folder, capture_output=True, text=True,
    )
    print(result.stdout.strip())

print("== mypy ==")
mypy("--strict")
print("== one mistake added on purpose ==")
module.write_text(module.read_text(encoding="utf-8") + "print(average([1.5, 2.5]))" +
chr(10), encoding="utf-8")
mypy("--strict")
print("== running the module ==")
run = subprocess.run([sys.executable, "field.py"], cwd=folder, capture_output=True,
text=True)
```

```
print(run.stdout.strip())
print("error while running:", run.stderr.strip().splitlines()[-1])
```

```
== mypy ==
Success: no issues found in 1 source file
== one mistake added on purpose ==
field.py:26: error: List item 0 has incompatible type "float"; expected "Record"
[list-item]
field.py:26: error: List item 1 has incompatible type "float"; expected "Record"
[list-item]
Found 2 errors in 1 file (checked 1 source file)
== running the module ==
519.25
None
error while running: TypeError: 'float' object is not subscriptable
```

Follow along:

- Mypy `--strict` declares the module clean: `Success: no issues found in 1 source file`. `cast(Sector, sector)` tells mypy that we are sure this text is one of the `Sector` values (mypy cannot know that from free text). That is our promise, not a check; the real check comes in Chapter 34.
- After the line `print(average([1.5, 2.5]))` is inserted (it should be given `Records`, not numbers), mypy reports two errors, one for each list element. The program has not been run at that point.
- If the module is run anyway, `519.25` and `None` from the first two lines are printed, then the program crashes at the inserted line with `TypeError: 'float' object is not subscriptable`. Mypy caught the same mistake **without** waiting for the program to crash.

Exercises

1. Add type hints to a function `average_weight(weights)` that takes a `list[float]` and returns a `float`.
2. Write a function `find_tree(items: list[str], name: str) -> int | None` that returns the position of `name` in the list, or `None` if absent.
3. Change `05_run_mypy.py`: fix the four mistakes in the example file, run again, and see the `Success` message.
4. **Challenge:** make a `TypedDict` called `Livestock` (keys `barn: str`, `kind: Literal["cattle", "goat", "chicken"]`, `weight_kg: float`), then a function `total_weight(data: list[Livestock]) -> float`. Check with mypy, including one record whose `kind` has a typo.

Chapter Summary

- A **type hint** is a type label on variables and functions; Python **does not check it** while running.
- Tools such as **mypy** (and `pyright`) check types **before** the program runs and report mismatches.
- Common types: `list[float]`, `dict[str, float]`, `float | None`, `Callable`, `Iterator`; **generics** (`[T]`) make types follow the content.
- `TypedDict`, `Literal`, and `Protocol` state the shape of a dict, limited choices, and “anything that has this”.
- Python 3.14 defers the evaluation of annotations; `annotationlib` reads them in the `VALUE`, `FORWARDREF`, and `STRING` formats.
- Project I4: the record-reading code is typed and passes `mypy --strict`; the inserted mistake is visible before running.

Next, in Chapter 34: data validation and configuration with `Pydantic`, a doorkeeper that works while the program runs.

Chapter 34 — Data Validation and Configuration: A Doorkeeper Called Pydantic

Series I4 — Advanced Python Language

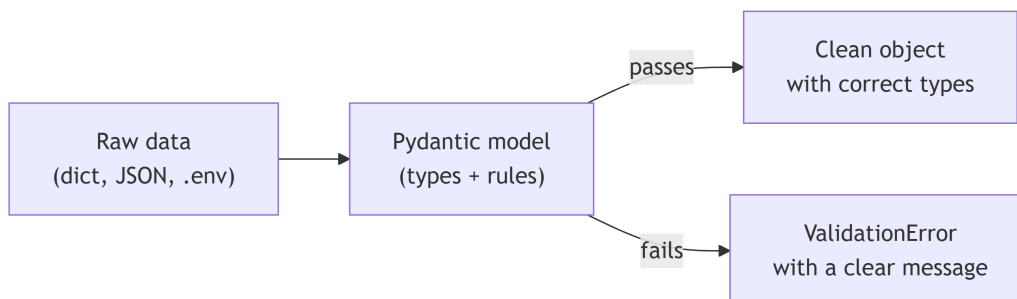
In this chapter, you will:

- Understand why data from outside must be **validated**, and the limits of a dataclass
- Build **Pydantic models**: validation, type conversion, and error messages
- Add rules with `Field`, `field_validator`, and `model_validator`
- Read app settings from environment variables and `.env` (`pydantic-settings`)
- Validate field data from five sectors at once: the Series I4 project

All scripts for this chapter are in the `code/ch34/` folder. On the web version, you can also [download them all at once \(zip\)](#).

In Chapter 33, type hints acted as labels read by a checking tool. But while the program runs, those labels guard nothing. Yet field data often comes from outside: forms, CSV files, JSON from an API, a clerk's typing. It can have typos, be empty, or be beyond common sense (water pH 17, a negative tree diameter).

A **doorkeeper** is needed: a checker that works **while the program runs**, rejects bad data with a clear message, and tidies data that is still acceptable. In Python, the most popular doorkeeper is **Pydantic**.



Pydantic as a doorkeeper: raw data is checked, then becomes a clean object or is rejected with an error message.

A dataclass is tidy but does not guard

In Chapter 16 you met `dataclass`: a compact way to make a class that holds data. See what happens when the content makes no sense:

```
# Chapter 34 - A dataclass is tidy, but does not check the content
from dataclasses import dataclass

@dataclass
class Tree:
    species: str
    diameter_cm: float
    healthy: bool
```

```

good = Tree("Teak", 32.5, True)
print(good)

messy = Tree("Teak", "thirty", "maybe") # no complaint at all
print(messy)
print(type(messy.diameter_cm).__name__)

```

```

Tree(species='Teak', diameter_cm=32.5, healthy=True)
Tree(species='Teak', diameter_cm='thirty', healthy='maybe')
str

```

The object `messy` is created without complaint: the `diameter` holds the text `'thirty'`, `healthy` holds `'maybe'`. Its type is `str`, although we wrote `float`. A `dataclass` only tidies up writing the class; it **does not check the content**. The mistake is felt later, perhaps far from its source, when that number gets calculated.

A Pydantic model

Pydantic is a data-validation library that uses type hints as rules. The version installed on the author's computer has major version 2 (the script prints only the "major version", because the full number changes over time). This book uses the Pydantic 2 style.

```

# Chapter 34 - A Pydantic model: checks and converts types
import pydantic
from pydantic import BaseModel

print("Pydantic major version:", pydantic.VERSION.split(".")[0])

class Tree(BaseModel):
    species: str
    diameter_cm: float
    healthy: bool = True # default value

t = Tree(species="Teak", diameter_cm="32.5", healthy="true") # text becomes number
and bool
print(t)
print(type(t.diameter_cm).__name__, type(t.healthy).__name__)
print(t.model_dump())

t2 = Tree(species="Meranti", diameter_cm=45) # int becomes float
print(t2.diameter_cm, t2.healthy)

```

```

Pydantic major version: 2
species='Teak' diameter_cm=32.5 healthy=True
float bool
{'species': 'Teak', 'diameter_cm': 32.5, 'healthy': True}
45.0 True

```

- A model is made as a subclass of `BaseModel`, and each attribute is given a type. `healthy: bool = True` has a default value.
- Pydantic **converts** when it makes sense: the text `"32.5"` becomes the `float` `32.5`, the text `"true"` becomes `True`, and the whole number `45` becomes `45.0`.
- The result is an object with correct types: `float` and `bool` (not `str`).
- `model_dump()` turns the object back into an ordinary dict.

There is a small lesson here. When I tried to fill `healthy="ya"` (the Indonesian word for “yes”), Pydantic **rejected it** (`bool_parsing`): it does not know that word. What it knows are standard forms such as `"true"`. Pydantic’s conversion is limited and documented, not guessing.

Clear error messages

When data cannot be accepted, Pydantic raises a `ValidationError` that explains **all** the problems at once, not only the first.

```
# Chapter 34 - Clear error messages
from pydantic import BaseModel, ValidationError

class Tree(BaseModel):
    species: str
    diameter_cm: float
    healthy: bool

try:
    Tree(species="Teak", diameter_cm="thirty", healthy="maybe")
except ValidationError as e:
    print("number of problems:", e.error_count())
    for problem in e.errors(include_url=False, include_input=False):
        print(problem["loc"], "|", problem["type"], "|", problem["msg"])

try:
    Tree(species="Teak")
except ValidationError as e:
    for problem in e.errors(include_url=False, include_input=False):
        print(problem["loc"], "|", problem["type"], "|", problem["msg"])
```

```
number of problems: 2
('diameter_cm',) | float_parsing | Input should be a valid number, unable to parse
string as a number
('healthy',) | bool_parsing | Input should be a valid boolean, unable to interpret input
('diameter_cm',) | missing | Field required
('healthy',) | missing | Field required
```

In the first attempt there are **two** problems: `diameter_cm` cannot be read as a number (`float_parsing`) and `healthy` cannot be read as yes/no (`bool_parsing`). In the second attempt, two required fields are not filled in (`missing`: “Field required”). Each problem has:

- `loc`: where the problem is (the field name),
- `type`: a stable error category that a program can process,
- `msg`: an explanatory sentence for humans.

The options `include_url=False` and `include_input=False` remove the link and the copy of the input from the display, so the output is short and stable.

Extra rules: Field and validators

A type alone is not enough. “A number” does not yet mean “sensible”. Extra rules can be attached with `Field` (simple limits) and validators (your own rules).

```
# Chapter 34 - Rules through Field and validators
from typing import Literal

from pydantic import BaseModel, Field, ValidationError, field_validator,
model_validator
```

```

class Fish(BaseModel):
    pond: str = Field(min_length=1, max_length=3)
    species: Literal["catfish", "tilapia"]
    count: int = Field(gt=0)
    avg_weight_g: float = Field(ge=1, le=5000)
    water_ph: float = Field(ge=0, le=14)

    @field_validator("pond")
    @classmethod
    def upper_case(cls, value: str) -> str:
        return value.upper() # tidies up at the same time

    @model_validator(mode="after")
    def check_catfish_weight(self):
        if self.species == "catfish" and self.avg_weight_g > 2000:
            raise ValueError("an average catfish weight above 2000 g is not realistic")
        return self

print(Fish(pond="a", species="catfish", count=800, avg_weight_g=540.5, water_ph=7.2))

for data in [
    dict(pond="B", species="carp", count=100, avg_weight_g=300, water_ph=7),
    dict(pond="B", species="tilapia", count=0, avg_weight_g=300, water_ph=15),
    dict(pond="C", species="catfish", count=10, avg_weight_g=3000, water_ph=7),
]:
    try:
        Fish(**data)
    except ValidationError as e:
        for m in e.errors(include_url=False, include_input=False):
            print(m["loc"], "|", m["msg"])
        print("--")

```

```

pond='A' species='catfish' count=800 avg_weight_g=540.5 water_ph=7.2
('species',) | Input should be 'catfish' or 'tilapia'
--
('count',) | Input should be greater than 0
('water_ph',) | Input should be less than or equal to 14
--
() | Value error, an average catfish weight above 2000 g is not realistic
--

```

- `Field(gt=0)` means “greater than 0”, `ge/le` mean “greater or equal”/“less or equal”, and `min_length/max_length` limit text length.
- `Literal["catfish", "tilapia"]` limits the fish species. "carp" is rejected: “Input should be ‘catfish’ or ‘tilapia’”.
- `@field_validator("pond")` checks (and may tidy) one field: the pond name "a" becomes "A".
- `@model_validator(mode="after")` checks a **combination** of fields after all are valid: catfish with an average weight above 2000 g is rejected, and the error location is `()` (empty) because the problem belongs to the whole model, not one field.

⚠ WARNING

Limits such as `avg_weight_g <= 5000` or “catfish no more than 2000 g” in this script are only **sample numbers for practice**, not farming advice. Set the real limits together with an extension officer or an experienced fish farmer.

App settings: environment variables and .env

An application has settings that differ on each computer: file locations, download limits, test mode, service keys. Putting them inside the code is a nuisance and, for secret keys, dangerous. A common practice: keep them in **environment variables** (settings outside the program) or in a **.env** file (a list of **NAME=value**), and read them when the program starts.

The **pydantic-settings** package reads settings into a class that is validated like a Pydantic model. This script uses **fake** values (no real secrets) and writes a sample **.env** file into **_tmp/**.

```
# Chapter 34 - App settings from environment variables and a .env file (fake values)
import os
from pathlib import Path

from pydantic import SecretStr
from pydantic_settings import BaseSettings, SettingsConfigDict

folder = Path(__file__).parent / "_tmp"
folder.mkdir(exist_ok=True)
env_file = folder / "example.env"
env_file.write_text("FIELD_UNIT_NAME=Example Pond\nFIELD_API_TOKEN=fake-token-not-a-secret\n"
                    "FIELD_DOWNLOAD_LIMIT=5\n", encoding="utf-8")

class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_prefix="FIELD_", env_file=env_file,
    env_file_encoding="utf-8")

    unit_name: str = "No name"
    download_limit: int = 10
    api_token: SecretStr = SecretStr("")
    test_mode: bool = False

s = Settings()
print("from .env      :", s.unit_name, s.download_limit, s.test_mode)
print("token (masked):", s.api_token)
print("token (opened on purpose):", s.api_token.get_secret_value() == "fake-token-not-a-secret")

os.environ["FIELD_DOWNLOAD_LIMIT"] = "25"          # an environment variable beats .env
os.environ["FIELD_TEST_MODE"] = "true"
s2 = Settings()
print("env beats .env:", s2.download_limit, s2.test_mode)

print("a direct argument is strongest:", Settings(download_limit=1).download_limit)
```

```
from .env      : Example Pond 5 False
token (masked): *****
token (opened on purpose): True
env beats .env: 25 True
a direct argument is strongest: 1
```

- `env_prefix="FIELD_"` means the setting `download_limit` is read from `FIELD_DOWNLOAD_LIMIT`. The prefix prevents clashes with variables of other programs.
- Values from **.env** are converted automatically: "5" becomes the `int` 5.
- **SecretStr** masks the value when printed (`*****`), so keys do not leak into logs. The real value opens only through `get_secret_value()`, which we use here only to compare.
- The order of strength, according to the pydantic-settings documentation: **direct argument** > **environment variable** > **.env file** > **default value**. The script proves it: **.env** gives 5, the environment variable overrides it to 25, and the argument `Settings(download_limit=1)` beats them all.

TIP

A `.env` file holds personal and often secret settings. **Do not** keep it in Git (add it to `.gitignore`, Chapter 20), and provide a sample file (`.env.example`) without secret values. In this book all values are fake.

JSON, lists of models, and schemas

```
# Chapter 34 - JSON, lists of models, and JSON Schema
import json

from pydantic import BaseModel, TypeAdapter

class Record(BaseModel):
    sector: str
    unit: str
    value: float

raw = '{"sector": "pond", "unit": "A", "value": "540.5"}'
r = Record.model_validate_json(raw)          # JSON text -> validated object
print(r)
print(r.model_dump_json())

records = TypeAdapter(list[Record]).validate_python([
    {"sector": "forest", "unit": "F1", "value": 32.5},
    {"sector": "orchard", "unit": "O2", "value": "245.5"},
])
print(len(records), records[1].value)

schema = Record.model_json_schema()          # basis of API docs and AI agent tools
print(json.dumps(schema, indent=2))
```

```
sector='pond' unit='A' value=540.5
{"sector": "pond", "unit": "A", "value": 540.5}
2 245.5
{
  "properties": {
    "sector": {
      "title": "Sector",
      "type": "string"
    },
    "unit": {
      "title": "Unit",
      "type": "string"
    },
    "value": {
      "title": "Value",
      "type": "number"
    }
  },
  "required": [
    "sector",
    "unit",
    "value"
  ],
```

```

    "title": "Record",
    "type": "object"
}

```

- `model_validate_json` validates JSON text directly ("540.5" becomes a number). `model_dump_json()` turns the object back into JSON.
- `TypeAdapter(list[Record])` validates a **list** of models at once.
- `model_json_schema()` produces a **JSON Schema**: a standard description of the shape of the data (fields, types, which are required). This schema is the basis of an API's automatic documentation.

Dataclass or Pydantic?

	<code>dataclass</code>	Pydantic <code>BaseModel</code>
Checks content while running	no	yes
Converts types ("32.5" to 32.5)	no	yes
Detailed error messages	no	yes
Part of the standard library	yes	no (extra library)
Suits	data inside a program that is already trusted	data coming in from outside

Rule of thumb: data crossing the "border" (forms, files, APIs, settings) is validated with Pydantic; clean internal data may be a `dataclass`.

The link to FastAPI and AI agents

This checking at the border is the basis of the next two topics in the book. **FastAPI** (Chapter 36), a web framework, uses Pydantic: the FastAPI package installed on the author's computer lists `pydantic` as a dependency (result of `pip show fastapi`), and in Chapter 36 you will see for yourself Pydantic models checking API input (in this chapter that is not run). In the discussion of AI agents at the Advanced level, similar schemas are used to describe the shape of data to a language model; that part has not been tested in this chapter.

Running Project I4: validating field data from five sectors

Field data from the five sectors has different shapes: trees have a diameter, the orchard has a yield, the ricefield has grain, the pond has weight and pH, the barn has livestock weight. We build five models, then one "gate" that picks the model by reading the `sector` field (called a *discriminated union*).

```

# Chapter 34 - Running project I4: validating field data from five sectors
from typing import Annotated, Literal

from pydantic import BaseModel, Field, TypeAdapter, ValidationError


class Forest(BaseModel):
    sector: Literal["forest"]
    unit: str
    diameter_cm: float = Field(gt=0, le=300)


class Orchard(BaseModel):
    sector: Literal["orchard"]
    unit: str
    yield_kg: float = Field(ge=0)

```

```

class Ricefield(BaseModel):
    sector: Literal["ricefield"]
    unit: str
    grain_kg: float = Field(ge=0)

class Pond(BaseModel):
    sector: Literal["pond"]
    unit: str
    avg_weight_g: float = Field(gt=0)
    water_ph: float = Field(ge=0, le=14)

class Barn(BaseModel):
    sector: Literal["barn"]
    unit: str
    weight_kg: float = Field(gt=0)

FieldData = Annotated[Forest | Orchard | Ricefield | Pond | Barn,
Field(discriminator="sector")]
checker = TypeAdapter(FieldData)

raw = [
    {"sector": "forest", "unit": "F1", "diameter_cm": "32.5"},
    {"sector": "orchard", "unit": "O1", "yield_kg": 245.5},
    {"sector": "ricefield", "unit": "R1", "grain_kg": -10},
    {"sector": "pond", "unit": "A", "avg_weight_g": 540.5, "water_ph": 7.2},
    {"sector": "pond", "unit": "B", "avg_weight_g": 498, "water_ph": 17},
    {"sector": "barn", "unit": "D1", "weight_kg": 412.5},
    {"sector": "swamp", "unit": "E1", "avg_weight_g": 100},
    {"sector": "barn", "unit": "D2"},
]

valid, rejected = [], []
for number, row in enumerate(raw, start=1):
    try:
        valid.append(checker.validate_python(row))
    except ValidationError as e:
        problem = e.errors(include_url=False, include_input=False)[0]
        rejected.append((number, ".".join(str(x) for x in problem["loc"]) or "sector",
        problem["msg"]))

print("valid:", len(valid), "| rejected:", len(rejected))
print("valid object types:", sorted({type(o).__name__ for o in valid}))
for number, place, reason in rejected:
    print(f"row {number}: {place} -> {reason}")

```

```

valid: 4 | rejected: 4
valid object types: ['Barn', 'Forest', 'Orchard', 'Pond']
row 3: ricefield.grain_kg -> Input should be greater than or equal to 0
row 5: pond.water_ph -> Input should be less than or equal to 14
row 7: sector -> Input tag 'swamp' found using 'sector' does not match any of the
expected tags: 'forest', 'orchard', 'ricefield', 'pond', 'barn'
row 8: barn.weight_kg -> Field required

```

Eight raw rows go in, **four pass** and **four are rejected**:

- Row 1: diameter_cm is the text "32.5", converted and accepted as a Forest.
- Row 3: grain_kg is -10, rejected because it must be >= 0.

- Row 5: pH 17, rejected because it is above 14.
- Row 7: the sector "swamp" is not one of the five, rejected, and the message names the valid choices.
- Row 8: the field `weight_kg` is missing ("Field required").
- Object types that passed: `Barn`, `Forest`, `Orchard`, `Pond`, each with a different shape yet all correctly typed.

Rejected data does not stop the program: it is recorded with its reason, and the rest is processed. That is the safe way of working with dirty field data.

Exercises

1. Make a model `Livestock` with fields `barn: str`, `kind: Literal["cattle", "goat", "chicken"]`, `age_months: int` (must be ≥ 0), and `weight_kg: float` (must be > 0). Try one correct record and one wrong one.
2. Change `03_error_messages.py`: give `diameter_cm=-5` to a model that has `Field(gt=0)`. Read the error message.
3. In `05_settings_env.py`, add a setting `output_folder: str` with default "results", then override it through an environment variable and through `.env`. See which one wins.
4. **Challenge:** add a sixth model `Equipment` to `07_project_five_sector_validation.py` (with `sector` equal to "tool"), put it in the union, and add two raw rows (one correct, one wrong). Recount the passed and rejected rows.

Chapter Summary

- Data from outside must be **validated while the program runs**; type hints alone guard nothing.
- A `dataclass` does not check content; a Pydantic `BaseModel` checks, converts, and gives per-field error messages.
- Extra rules: `Field` (limits), `field_validator` (one field), `model_validator` (combined fields).
- **pydantic-settings** reads settings from environment variables and `.env`; order of strength: argument $>$ environment $>$ `.env` $>$ default; `SecretStr` masks secrets.
- `TypeAdapter`, `model_validate_json`, and `model_json_schema` validate lists, JSON, and produce schemas.
- Project I4: five-sector data is validated through a discriminated union; four pass, four are rejected with reasons.

Next, in Chapter 35: async and concurrency, a way to do many waiting jobs at once, and the close of Series I4.

Chapter 35 — Async and Concurrency: Doing Many Things at Once

Series I4 — Advanced Python Language

In this chapter, you will:

- Tell **waiting** jobs (I/O) apart from **calculating** jobs (CPU)
- Write `async/await` code with `asyncio` (`gather`, `TaskGroup`, `timeout`, `Semaphore`)
- Use **threads** and **processes** through `concurrent.futures`
- Understand the GIL and Python 3.14's *free-threaded* mode, and when to use which
- Download lots of weather data at once from a local fake server: the peak of the Series I4 project

All scripts for this chapter are in the `code/ch35/` folder. On the web version, you can also [download them all at once](#) ([zip](#)).

Imagine you fetch rainfall data from 20 stations. Each network request needs waiting time, say half a second. Done one by one: $20 \times 0.5 = 10$ seconds, and almost all of it is spent **waiting**. Like a farmer watering 20 plots with one hose, waiting for the water to reach each plot before moving to the next. Yet while waiting, his hands could be tending another plot.

That is the heart of this chapter: arranging things so that waiting time is not wasted.

Two kinds of jobs

Kind	Examples	What holds it back
I/O-bound (mostly waiting)	download data, read a disk, ask a database	waiting for network or disk
CPU-bound (mostly calculating)	simulations, heavy number crunching	processor speed

The choice of tool depends on this kind. The three main tools:

Three ways to do many things at once

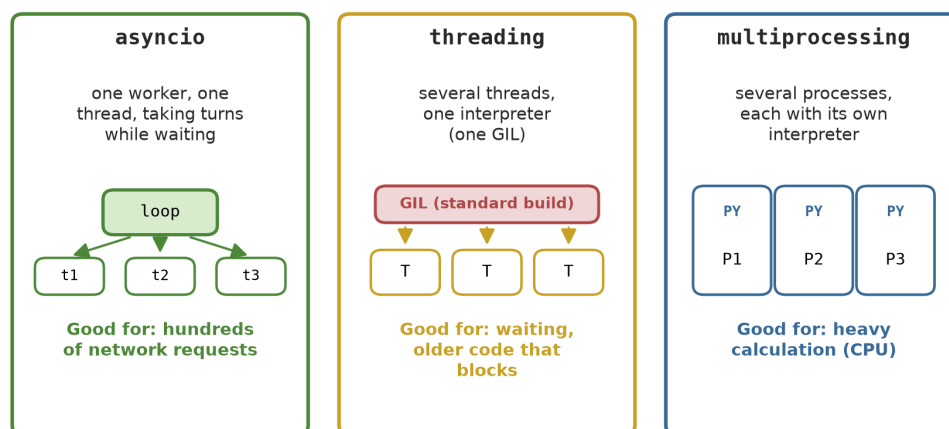


Figure 0.1: Three ways to do many things at once: asyncio, threading, multiprocessing.

asyncio: one worker who is good at taking turns

An ordinary function runs from start to finish. An **async def** function (called a **coroutine**) can **pause** at an **await** point and hand the turn to other jobs that are waiting. One worker (one thread, called the **event loop**) manages the turns.

```
# Chapter 35 - Async basics: coroutine, await, and asyncio.run
import asyncio

async def measure_ph(pond):
    await asyncio.sleep(0.01)      # waits without blocking other work
    return f"pond {pond} pH = 7.2"

# Calling an async function does NOT run it: it only creates a coroutine object
obj = measure_ph("A")
print(type(obj).__name__)
obj.close()                       # thrown away without running

async def main():
    result = await measure_ph("A") # await: run it and wait for the result
    print(result)

asyncio.run(main())               # entry point: starts the event loop
```

```
coroutine
pond A pH = 7.2
```

- Calling `measure_ph("A")` **does not run it**; it only creates an object of type `coroutine`. (In the script, that object is closed straight away with `close()`.) Forgetting `await` is a very common beginner mistake, and usually produces only a warning.
- `await measure_ph("A")` runs it and waits for the result.
- `asyncio.run(main())` is the entry point that switches on the event loop. It is usually called **once**, at the outermost part of the program.
- `asyncio.sleep(0.01)` is a **non-blocking** wait. Compare with ordinary `time.sleep()`, which freezes the whole program.

gather: many waiting jobs together

```
# Chapter 35 - Many waiting jobs at once with gather
import asyncio

finished = []

async def download(name, seconds):
    await asyncio.sleep(seconds)
    finished.append(name)
    return name, seconds

async def one_after_another():
    for name, seconds in [("S1", 0.06), ("S2", 0.02), ("S3", 0.04)]:
        await download(name, seconds)
```

```

async def all_together():
    return await asyncio.gather(download("S1", 0.06), download("S2", 0.02),
download("S3", 0.04))

async def main():
    loop = asyncio.get_running_loop()
    t0 = loop.time()
    await one_after_another()
    sequential_time = loop.time() - t0
    print("finishing order (one after another):", finished)

    finished.clear()
    t0 = loop.time()
    results = await all_together()
    concurrent_time = loop.time() - t0
    print("finishing order (together):", finished)
    print("gather results follow the written order:", [n for n, _ in results])
    print("together is faster than 70% of the sequential time:", concurrent_time < 0.7
* sequential_time)

asyncio.run(main())

```

```

finishing order (one after another): ['S1', 'S2', 'S3']
finishing order (together): ['S2', 'S3', 'S1']
gather results follow the written order: ['S1', 'S2', 'S3']
together is faster than 70% of the sequential time: True

```

Three fake downloads with waiting times of 0.06, 0.02, and 0.04 seconds:

- **One after another:** finishing in the order they were run: S1, S2, S3.
- **Together (gather):** finishing in order of **waiting time**: S2 (0.02), S3 (0.04), then S1 (0.06). The short one does not have to wait for the long one.
- The **order of results** from **gather** still follows the **written** order (S1, S2, S3), whatever the finishing order. This is documented and convenient: result number *n* always belongs to job number *n*.
- The script does not print raw times (they change on every run), only a comparison: the concurrent time is under 70% of the sequential time (**True**). The sequential total is about 0.12 seconds, the concurrent about 0.06 seconds.

TaskGroup, time limits, and a concurrency cap

```

# Chapter 35 - TaskGroup, time limits, and a cap on concurrency (Semaphore)
import asyncio

active = 0
peak = 0

async def work(number, semaphore):
    global active, peak
    async with semaphore:          # at most 2 inside at the same time
        active += 1
        peak = max(peak, active)
        await asyncio.sleep(0.02)
        active -= 1
    return number * 10

```



```

async def slow():
    await asyncio.sleep(1)

async def main():
    semaphore = asyncio.Semaphore(2)
    async with asyncio.TaskGroup() as group:
        tasks = [group.create_task(work(i, semaphore)) for i in range(1, 7)]
        print("results:", [t.result() for t in tasks])
        print("most running at the same time:", peak)

    try:
        async with asyncio.timeout(0.05):
            await slow()
    except TimeoutError:
        print("cancelled: time limit exceeded")

asyncio.run(main())

```

```

results: [10, 20, 30, 40, 50, 60]
most running at the same time: 2
cancelled: time limit exceeded

```

- **asyncio.TaskGroup** (Python 3.11+) starts a group of tasks and **waits for all of them** when the **async** with block ends. If one task fails, the others are cancelled: safer than a bare **gather**.
- **asyncio.Semaphore(2)** lets only two tasks into its inside at once. The proof: of six tasks, the highest peak running together is 2. This matters so a server is not flooded with hundreds of requests at once.
- **asyncio.timeout(0.05)** (Python 3.11+) cancels work that exceeds the limit, then raises **TimeoutError**. The job **slow()**, which needs 1 second, is stopped after 0.05 seconds.

TIP

Always give network jobs a **time limit** (**timeout**) and a **count limit** (**Semaphore**). An outside server may be slow or stuck, and sending 1,000 requests at once can look like an attack.

Threads: for waiting jobs in ordinary code

What if your function is not **async** (for example an older library that blocks)? Use **threads**: several lines of work inside one program. The easiest way: **ThreadPoolExecutor**.

```

# Chapter 35 - Threads for waiting jobs (I/O) with the standard library
import threading
import time
from concurrent.futures import ThreadPoolExecutor

def read_sensor(name):
    time.sleep(0.05)                # pretends to wait for network or disk
    return name, len(name) * 1.5

with ThreadPoolExecutor(max_workers=4) as pool:
    results = list(pool.map(read_sensor, ["temp", "ph", "oxygen", "murk"]))
    print(results)                  # map keeps the input order

# Data shared between threads must be protected by a Lock
total = 0

```

```
lock = threading.Lock()

def add_many():
    global total
    for _ in range(100_000):
        with lock:
            total += 1

threads = [threading.Thread(target=add_many) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("total with Lock:", total)
```

```
[('temp', 6.0), ('ph', 3.0), ('oxygen', 9.0), ('murk', 6.0)]
total with Lock: 400000
```

- `pool.map(...)` runs `read_sensor` on four threads and **keeps the result order** the same as the input: `temp`, `ph`, `oxygen`, `murk`. That makes the output stable.
- In the lower part, four threads add to `total` together. Without protection, two threads can read the old value at the same time and one of the additions is lost (a *race condition*). With a **Lock**, only one thread enters at a time: the total is exactly 400,000 (4 x 100,000).
- To run an ordinary blocking function from inside `async` code, there is `asyncio.to_thread(...)` (documented since Python 3.9).

Processes: for calculating jobs

```
# Chapter 35 - Separate processes for calculating jobs (CPU)
from concurrent.futures import ProcessPoolExecutor

def heavy_sum(n):
    """Pure CPU work: sum of squares 1..n."""
    return sum(i * i for i in range(1, n + 1))

if __name__ == "__main__":
    # REQUIRED on Windows: every child process
    imports this file
    jobs = [200_000, 300_000, 400_000, 500_000]
    with ProcessPoolExecutor(max_workers=4) as pool:
        results = list(pool.map(heavy_sum, jobs))
    for n, r in zip(jobs, results):
        print(n, r)
```

```
200000 2666686666700000
300000 9000045000050000
400000 21333413333400000
500000 41666791666750000
```

`ProcessPoolExecutor` runs the function in **separate processes**: each process is its own Python program with its own memory. Results are sent back to the main program, and `map` keeps their order (200,000, 300,000, 400,000, 500,000).

⚠ WARNING

On **Windows**, code that creates new processes **must** sit under `if __name__ == "__main__":`. The reason: Windows starts child processes with the *spawn* method, which launches a fresh interpreter that **imports** your file from the top. Without that guard, every child process would repeat the process-creating code and raise a `RuntimeError`. The multiprocessing documentation names *spawn* as the default method on Windows and macOS, and recommends this guard (“safe importing of main module”). The scripts in this chapter always use it when creating processes. Because functions and data are sent between processes, both must be **picklable** (serialisable); functions must be defined at the top level of a module, not as lambdas (the author tried a lambda and got a `PicklingError`).

The GIL: why threads do not always speed things up

Standard Python has a **GIL** (*Global Interpreter Lock*), a lock that makes sure only **one thread** runs Python code at a time within one process. As a result, threads **do not speed up CPU work**; threads are useful for waiting jobs because while one thread waits, others can run. Let us prove it on this computer:

```
# Chapter 35 - GIL: threads do not speed up CPU work (standard build), processes do
import sys
import time
from concurrent.futures import ProcessPoolExecutor, ThreadPoolExecutor

def heavy_sum(n):
    return sum(i * i for i in range(n))

def measure(function):
    t0 = time.perf_counter()
    function()
    return time.perf_counter() - t0

if __name__ == "__main__":
    print("GIL enabled in this Python:", sys._is_gil_enabled())
    load = [5_000_000] * 4

    sequential = measure(lambda: [heavy_sum(n) for n in load])
    with ThreadPoolExecutor(max_workers=4) as pool:
        threaded = measure(lambda: list(pool.map(heavy_sum, load)))
    with ProcessPoolExecutor(max_workers=4) as pool:
        processes = measure(lambda: list(pool.map(heavy_sum, load)))

    print("4 threads speed it up by more than 1.5x:", sequential / threaded > 1.5)
    print("4 processes speed it up by more than 1.3x:", sequential / processes > 1.3)
```

```
GIL enabled in this Python: True
4 threads speed it up by more than 1.5x: False
4 processes speed it up by more than 1.3x: True
```

- `sys._is_gil_enabled()` reports whether the GIL is active in the Python running the script: `True`.
- Four threads do **not** speed up heavy calculation by more than 1.5 times (`False`): the lock makes them take turns.
- Four processes speed it up by more than 1.3 times (`True`), because each process has its own interpreter and GIL, so they truly run in parallel.
- As before, the script prints only comparisons, not seconds. These `True/False` results were checked repeatedly on the author’s computer; on yours the exact numbers will differ (for example with fewer processor cores), so the 1.3 threshold may not be reached on a computer with few cores.

The *free-threaded* mode

Since Python 3.13 there is an optional **free-threaded** build with the GIL switched off, so threads can run in parallel on many cores. According to the documentation, Python 3.14 declares official support for this mode (the PEP 779 reference in “What’s New”), but it is **not the default**: it must be chosen when installing or built separately, and not all C libraries are ready for it. How to check: `sys._is_gil_enabled()` and `sysconfig.get_config_var("Py_GIL_DISABLED")`. On the author’s computer only the standard build (GIL active) was tested; a free-threaded build was **not tested** in this chapter.

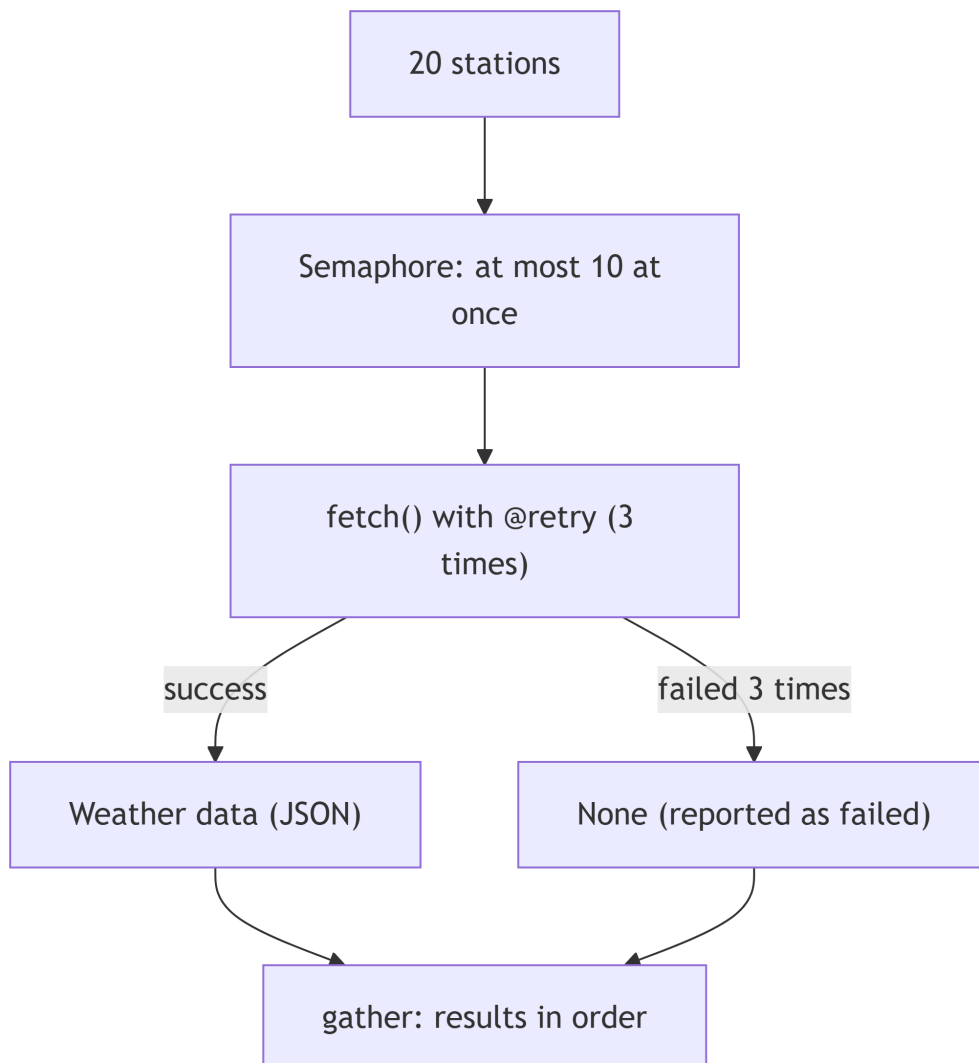
i REMEMBER

A guide to choosing:

Your problem	Choice	Reason
Hundreds of network requests, library supports <code>async</code>	asyncio	light, one thread, easy to limit
Waiting, but an older library blocks	threads (<code>ThreadPoolExecutor</code>)	no need to rewrite as <code>async</code>
Pure heavy calculation (CPU)	processes (<code>ProcessPoolExecutor</code>)	gets around the GIL with many interpreters
Only a few simple jobs	none of these	ordinary code is easier to maintain

Running Project I4: downloading lots of weather data with `async`

The peak of the Series I4 project: fetching data from 20 weather/price stations at once. To keep it safe and stable, **the server is fake**: built on this computer itself with `aiohttp`, at the local address `127.0.0.1` on a port chosen by the system. **No internet** is used. The server imitates a slow network (0.02 seconds) and outages: station S07 fails once (status 503), S13 fails three times.



Project flow: 20 requests capped at 10 at a time, each request retried on failure, results collected in order.

```

# Chapter 35 - Running project I4: downloading many weather records from a FAKE
local server
import asyncio
import logging
import sys
from functools import wraps

import aiohttp
from aiohttp import web

logging.basicConfig(stream=sys.stdout, level=logging.WARNING, format="%(levelname)s
%(message)s")
log = logging.getLogger("download")

outages = {"S07": 1, "S13": 3}          # fake stations that fail a few times

# ----- fake server (only on this computer, no internet) -----

```

```

async def serve_data(request):
    station = request.match_info["station"]
    await asyncio.sleep(0.02)           # pretends the network is slow
    if outages.get(station, 0) > 0:
        outages[station] -= 1
        return web.Response(status=503, text="busy")
    number = int(station[1:])
    return web.json_response({"station": station, "rainfall_mm": number * 1.5,
                             "rice_price": 6000 + number * 10})

# ----- async client with retries (the Chapter 31 decorator, async version)
-----
def retry(maximum=3):
    def decorator(function):
        @wraps(function)
        async def wrapper(session, address, station):
            for attempt in range(1, maximum + 1):
                try:
                    return await function(session, address, station)
                except aiohttp.ClientResponseError as e:
                    log.warning("%s attempt %d: status %d", station, attempt, e.status)
                    if attempt == maximum:
                        raise
                    await asyncio.sleep(0.01)
            return wrapper
    return decorator

@retry(maximum=3)
async def fetch(session, address, station):
    async with session.get(f"{address}/data/{station}") as response:
        response.raise_for_status()
        return await response.json()

async def fetch_all(session, address, stations, limit):
    semaphore = asyncio.Semaphore(limit)

    async def one(station):
        async with semaphore:
            try:
                return await fetch(session, address, station)
            except aiohttp.ClientResponseError:
                return None

    return await asyncio.gather(*(one(s) for s in stations))

async def main():
    app = web.Application()
    app.add_routes([web.get("/data/{station}", serve_data)])
    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, "127.0.0.1", 0)           # port 0: the system picks a
free port
    await site.start()
    port = site._server.sockets[0].getsockname()[1]
    address = f"http://127.0.0.1:{port}"

```

```

stations = [f"S{n:02d}" for n in range(1, 21)]
async with aiohttp.ClientSession() as session:
    loop = asyncio.get_running_loop()
    t0 = loop.time()
    results = await fetch_all(session, address, stations, limit=10)
    elapsed = loop.time() - t0

    await runner.cleanup()
    succeeded = [r for r in results if r]
    failed = [s for s, r in zip(stations, results) if r is None]
    print("stations requested:", len(stations), "| succeeded:", len(succeeded), "|
failed:", failed)
    average = sum(r["rainfall_mm"] for r in succeeded) / len(succeeded)
    print("average rainfall (mm):", round(average, 2))
    print("highest rice price at station:", max(succeeded, key=lambda r:
r["rice_price"])["station"])
    print("faster than one by one (20 x 0.02 s = 0.4 s):", elapsed < 0.3)

if __name__ == "__main__":
    asyncio.run(main())

```

```

WARNING S07 attempt 1: status 503
WARNING S13 attempt 1: status 503
WARNING S13 attempt 2: status 503
WARNING S13 attempt 3: status 503
stations requested: 20 | succeeded: 19 | failed: ['S13']
average rainfall (mm): 15.55
highest rice price at station: S20
faster than one by one (20 x 0.02 s = 0.4 s): True

```

Follow the result:

- The `retry` decorator from Chapter 31 is used again, now in its `async` version (with `await asyncio.sleep` for the pause). This ties Chapters 31 and 35 together.
- The warning log shows: S07 failed on attempt 1 and succeeded on attempt 2 (no second warning), while S13 failed on attempts 1, 2, and 3, so it gave up and its result is `None`.
- The summary: 20 requested, **19 succeeded**, failed: `['S13']`. Average rainfall is computed only from the successful ones: 15.55 mm (fake numbers: station number n returns $n \times 1.5$). The highest rice price is at station S20.
- With `Semaphore(10)`, twenty requests finish far faster than one by one ($20 \times 0.02 = 0.4$ seconds): the comparison is `True`. The script prints no seconds.
- The server and the client run **in the same event loop**, so the whole experiment fits in one script.
- The server is shut down neatly with `runner.cleanup()`; the client session is closed by `async with`, an **asynchronous context manager** (the `async` relative of Chapter 32).

WARNING

This is a **simulation**. With a real API, follow the provider's rate limits and usage rules, and do not hit it with hundreds of requests without permission. The examples in this book contact no outside service.

Exercises

1. Change `02_gather_concurrent.py`: add a fourth station with a waiting time of 0.01 seconds. Predict the concurrent finishing order first, then run it and compare.
2. Change `Semaphore(2)` in `03_taskgroup_timeout.py` to 3. What peak is printed now? Why?
3. Change `limit=10` in `07_project_async_download.py` to `limit=1`. The summary results (succeeded and failed) stay the same; only the last line (the speed comparison) changes. Why?

4. **Challenge:** add a third broken station, `S18`, to `outages` with value 5 (fails five times), run again, and recompute the average rainfall by hand to check that it matches the output.

Chapter Summary

- Tell **waiting (I/O)** and **calculating (CPU)** jobs apart; the choice of tool depends on it.
- **asyncio**: `async def` + `await` + `asyncio.run`; `gather` keeps result order; `TaskGroup`, `timeout`, and `Semaphore` manage and limit tasks.
- **Threads** (`ThreadPoolExecutor`) suit waiting in ordinary code; shared data is protected with a `Lock`.
- **Processes** (`ProcessPoolExecutor`) suit calculating; on Windows if `__name__ == "__main__":` is required.
- The **GIL** keeps threads from speeding up CPU work in standard Python; the **free-threaded** build is a separate option.
- Project I4: 20 stations are downloaded at once from a fake server, with retrying and a count limit.

Closing Series I4

Congratulations! You have finished **Series I4** and this second volume of the Intermediate level. Look at what your code can do now compared with the start of the series:

- read big files **one line at a time** without burdening memory (generators, Chapter 30),
- wrap functions with **logging and retrying** without changing what is inside (decorators, Chapter 31),
- **open and close** resources with certainty, and write reports safely (context managers, Chapter 32),
- carry **type labels** and be checked by mypy before running (type hints, Chapter 33),
- reject bad field data with **clear messages** and read settings safely (Pydantic, Chapter 34),
- download lots of data **at once** with limits and retrying (async, Chapter 35).

The code is strong. Series I5 turns it into an application that other people can use: web services (Chapter 36), Python in the browser (Chapter 37), maps and spatial data (Chapters 38 to 40), automation and interfaces (Chapters 41 and 42), and dashboards (Chapter 43). The Pydantic and async you just learned will be used right away in FastAPI.

Next, in Chapter 36: web development, namely Flask, FastAPI, an introduction to Django, and REST APIs.

Chapter 36 — Web Development: Opening a Data Stall on the Internet

Series I5 — A First Application: Web, Maps, and Dashboards

In this chapter, you will:

- Understand how the web works: a **request** and a **response**
- Build a web app and a **REST API** with **Flask**
- Build a modern API with **FastAPI**, with Pydantic validation and automatic documentation
- Get a first look at **Django**, the “everything included” framework
- Test web apps without starting a server, then start a real server just for a moment
- Begin the farm unit data API: the Series I5 running project

All scripts for this chapter are in the `code/ch36/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Up to Chapter 35, your programs lived on one computer: your own. Yet data about gardens, ponds, and pens is often needed by other people: an extension officer in the office, a worker in the field, or an app on a phone. The most common way to share is to **open a data stall**: your program sits and waits for orders (“please give me pond A’s data”), then answers. That is a **web server** and an **API** (*Application Programming Interface*, the “door” other programs use to ask for data).

One request, one response

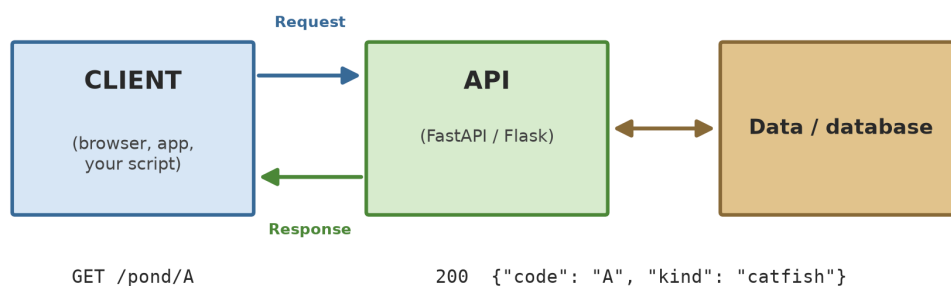


Figure 0.1: One request, one response: the basis of every web application.

How the web works in one paragraph

Every web interaction is an **order and an answer**. The client (a browser, an app, or your Python script) sends a **request** holding a *method* and an *address*, for example `GET /ponds/A`. The server replies with a **status code** and some content. The content is usually an HTML page or **JSON** data (text shaped like a dict, Chapter 14).

Method	Meaning	Example
GET	Ask for data	See the list of ponds
POST	Add new data	Record a new pond
PUT / PATCH	Change data	Update the fish count

Method	Meaning	Example
DELETE	Remove data	Delete a pond

Status code	Meaning
200	Success
201	Successfully created (after a POST)
404	Not found
409	Conflict (e.g. the code already exists)
422	The input failed validation

A **REST API** is just a habit for arranging addresses and methods: the address names a *thing* (`/ponds/A`), the method names an *action*.

i REMEMBER

The code in this chapter does **not start a server** to wait forever. We use a **test client** (a browser look-alike that calls the application directly from inside Python). The results are exactly the same, but fast, safe, and with fixed output. One script (07) starts a real server for a moment in a thread and then stops it.

Flask: the smallest stall

Flask is a light web framework: you start with a few lines and add things as you need them. Each address is connected to a function with the *decorator* `@app.get(...)` (from Chapter 31).

```
# Chapter 36 - Flask: the smallest web app, tested without starting a server
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.get("/")
def home():
    return "Hello from the farm!"

@app.get("/greet/<name>")
def greet(name):
    return {"message": f"Good morning, {name}"}

@app.get("/search")
def search():
    commodity = request.args.get("commodity", "all")
    return jsonify(searched=commodity)

client = app.test_client()
r = client.get("/")
print(r.status_code, r.content_type, r.get_data(as_text=True))
r = client.get("/greet/Budi")
print(r.status_code, r.content_type, r.get_json())
print(client.get("/search?commodity=coffee").get_json())
```

```
r = client.get("/does-not-exist")
print(r.status_code, r.status)
```

```
200 text/html; charset=utf-8 Hello from the farm!
200 application/json {'message': 'Good morning, Budi'}
{'searched': 'coffee'}
404 404 NOT FOUND
```

- "/" returns text, so the content type is `text/html`. Returning a **dict** automatically becomes JSON (`application/json`).
- `<name>` in the address is a **path parameter**: the value `Budi` enters the function as the variable `name`.
- `request.args` reads **query parameters**, the part after the `?` in the address.
- An address that is not registered gives `404 NOT FOUND` automatically.
- `app.test_client()` is the test client: it calls the app without opening a network port.

A REST API with Flask

Now a stall that can record things: the list of ponds is kept in a dict (in memory, so it vanishes when the program stops; lasting storage comes in later chapters).

```
# Chapter 36 - REST API with Flask: a list of ponds (kept in memory)
from flask import Flask, abort, jsonify, request

app = Flask(__name__)
ponds = {"A": {"kind": "catfish", "count": 800}, "B": {"kind": "tilapia", "count": 500}}

@app.get("/ponds")
def list_all():
    return jsonify(ponds)

@app.get("/ponds/<code>")
def one(code):
    if code not in ponds:
        abort(404, description=f"pond {code} does not exist")
    return jsonify(ponds[code])

@app.post("/ponds/<code>")
def add(code):
    body = request.get_json()           # the request body (JSON)
    ponds[code] = {"kind": body["kind"], "count": body["count"]}
    return jsonify(ponds[code]), 201    # 201 = Created

@app.delete("/ponds/<code>")
def remove(code):
    ponds.pop(code, None)
    return "", 204                      # 204 = success, no content

@app.errorhandler(404)
def not_found(error):
    return jsonify(error=error.description), 404

c = app.test_client()
print("GET /ponds ->", c.get("/ponds").status_code, sorted(c.get("/
```

```

ponds").get_json())
r = c.post("/ponds/C", json={"kind": "catfish", "count": 650})
print("POST /ponds/C ->", r.status_code, r.get_json())
print("GET /ponds/C ->", c.get("/ponds/C").get_json())
print("DELETE /ponds/C ->", c.delete("/ponds/C").status_code)
r = c.get("/ponds/C")
print("GET /ponds/C ->", r.status_code, r.get_json())

```

```

GET /ponds -> 200 ['A', 'B']
POST /ponds/C -> 201 {'count': 650, 'kind': 'catfish'}
GET /ponds/C -> {'count': 650, 'kind': 'catfish'}
DELETE /ponds/C -> 204
GET /ponds/C -> 404 {'error': 'pond C does not exist'}

```

Walk through the five output lines: the GET list gives 200 with codes A and B; the POST adds pond C and is answered **201**; the DELETE is answered **204** (success, no content); the GET for pond C after deletion is answered **404** with the message we wrote through `errorhandler`. Notice: Flask does **not check** the content of our POST. If `count` is missing, our code will stop with a `KeyError`. We would have to write that checking ourselves. This is where FastAPI shines.

FastAPI: data types that work for you

FastAPI uses *type hints* (Chapter 33) and **Pydantic** (Chapter 34) to check data automatically and write its own documentation. Functions can be ordinary (`def`) or **async** (Chapter 35).

```

# Chapter 36 - FastAPI: path parameters and query parameters with data types
from fastapi import FastAPI
from fastapi.testclient import TestClient

app = FastAPI()

@app.get("/")
def home():
    return {"message": "Hello from the farm!"}

@app.get("/ponds/{code}")
def pond(code: str, min_count: int = 0):          # code from the address; min_count
    from ?min_count=
    return {"code": code, "min_count": min_count}

@app.get("/harvest/{block}")
async def harvest(block: str):                    # async functions are fine (see
    Chapter 35)
    return {"block": block, "yield_kg": 245.5}

client = TestClient(app)
print(client.get("/").json())
print(client.get("/ponds/A?min_count=500").json())
print(client.get("/harvest/A").json())

# Data types are checked automatically: "many" is not an integer
r = client.get("/ponds/A?min_count=many")
print(r.status_code)
print(r.json()["detail"][0]["msg"])

```

```
{'message': 'Hello from the farm!'}
{'code': 'A', 'min_count': 500}
{'block': 'A', 'yield_kg': 245.5}
422
Input should be a valid integer, unable to parse string as an integer
```

`min_count: int = 0` means “an integer, default 0”. When we send `?min_count=many`, FastAPI **rejects** it with status 422 and a clear message (Input should be a valid integer...) without a single line of checking written by us.

A request body with a Pydantic model

```
# Chapter 36 - FastAPI + Pydantic: a validated request body
from fastapi import FastAPI, HTTPException
from fastapi.testclient import TestClient
from pydantic import BaseModel, Field

app = FastAPI()

class NewPond(BaseModel):
    kind: str
    count: int = Field(gt=0)
    water_ph: float = Field(ge=0, le=14)

class PondOut(BaseModel):
    code: str
    kind: str
    count: int
    # shape of the response

store: dict[str, NewPond] = {}

@app.post("/ponds/{code}", response_model=PondOut, status_code=201)
def add(code: str, pond: NewPond):
    store[code] = pond
    return {"code": code, **pond.model_dump()} # water_ph does not come out:
    # filtered by response_model

@app.get("/ponds/{code}", response_model=PondOut)
def read(code: str):
    if code not in store:
        raise HTTPException(status_code=404, detail=f"pond {code} does not exist")
    return {"code": code, **store[code].model_dump()}

c = TestClient(app)
r = c.post("/ponds/A", json={"kind": "catfish", "count": 800, "water_ph": 7.2})
print(r.status_code, r.json())
r = c.post("/ponds/B", json={"kind": "tilapia", "count": -5, "water_ph": 17})
print(r.status_code)
for error in r.json()["detail"]:
    print(" ", error["loc"], "->", error["msg"])
r = c.get("/ponds/Z")
print(r.status_code, r.json())
```

```

201 {'code': 'A', 'kind': 'catfish', 'count': 800}
422
    ['body', 'count'] -> Input should be greater than 0
    ['body', 'water_ph'] -> Input should be less than or equal to 14
404 {'detail': 'pond Z does not exist'}

```

- NewPond is a Pydantic model for **incoming** data: `count` must be greater than 0 and `water_ph` between 0 and 14.
- PondOut is the shape of the answer. `response_model` **filters** the answer: `water_ph` does not come out in the first output line. This is a safe way to keep internal data from leaking.
- For pond B, two rules are broken at once, and FastAPI reports **both**: `count` and `water_ph`, with their locations (`['body', 'count']`).
- `HTTPException` turns our problem into the right status code (404).

Documentation that writes itself

```

# Chapter 36 - FastAPI writes its own documentation (OpenAPI)
from fastapi import FastAPI
from fastapi.testclient import TestClient

app = FastAPI(title="Farm API", version="0.1")

@app.get("/ponds/{code}", summary="Get one pond", tags=["ponds"])
def pond(code: str):
    """Returns the data of one pond by its code."""
    return {"code": code}

@app.get("/harvest", summary="List harvests", tags=["harvest"])
def harvest(block: str | None = None):
    return []

c = TestClient(app)
schema = c.get("/openapi.json").json()
print("title:", schema["info"]["title"], "| version:", schema["info"]["version"])
for address, methods in schema["paths"].items():
    for name, detail in methods.items():
        print(f"{name.upper():4s} {address:15s} {detail['summary']}")
print("page /docs (Swagger UI) exists:", c.get("/docs").status_code == 200)
print("page /redoc exists:", c.get("/redoc").status_code == 200)

```

```

title: Farm API | version: 0.1
GET  /ponds/{code}    Get one pond
GET  /harvest         List harvests
page /docs (Swagger UI) exists: True
page /redoc exists: True

```

From the code alone, FastAPI makes the **OpenAPI schema** (the standard description of an API) at `/openapi.json`, plus the interactive documentation pages `/docs` (Swagger UI) and `/redoc`. An extension officer or a mobile developer can try your API from a browser without reading the code.

TIP

To open that documentation on your computer, run a real server in a terminal: `uvicorn file_name:app --reload`, then open `http://127.0.0.1:8000/docs`. We do not run it in this chapter because a server waits endlessly; that is a job for your terminal.

A first look at Django

Django is an “everything included” framework (*batteries included*): user accounts, an admin panel, a database layer (ORM), and HTML templates are all provided. It suits big sites full of pages and data. A real Django project is created with the command `django-admin startproject name`, which produces several files. We do not run that command here. As an introduction, here is a **one-file** Django project:

```
# Chapter 36 - Django: a one-file project (for introduction only)
import django
from django.conf import settings

settings.configure(
    DEBUG=False,
    SECRET_KEY="example-only-not-a-secret",
    ROOT_URLCONF=__name__,          # the address list lives in this file
    ALLOWED_HOSTS=["testserver"],
)
django.setup()

from django.http import JsonResponse
from django.test import Client
from django.urls import path

def home(request):
    return JsonResponse({"message": "Hello from Django"})

def pond(request, code):
    return JsonResponse({"code": code, "kind": request.GET.get("kind", "?")})

urlpatterns = [path("", home), path("ponds/<str:code>/", pond)]

client = Client()
print(client.get("/").json())
print(client.get("/ponds/A/?kind=catfish").json())
print(client.get("/does-not-exist/").status_code)

{'message': 'Hello from Django'}
{'code': 'A', 'kind': 'catfish'}
404
```

`settings.configure(...)` replaces the `settings.py` file; `urlpatterns` is the address list; `Client` is Django’s own test client. The `SECRET_KEY` here is just example text. In a real project, that key must be secret and read from an environment variable (Chapter 34).

	Flask	FastAPI	Django
Style	Micro, free-form	Modern, type-based	Everything included
Data validation	Write it yourself	Automatic (Pydantic)	Forms and models
API documentation	An add-on	Automatic	An add-on
Good for	Prototypes, small services	APIs for data and models	Full sites with an admin

We use **FastAPI** for the running project because our goal is a data API.

A real server, just for a moment

A test client does not travel through the network. To prove that the app can really be called over HTTP, the script below starts **uvicorn** (the server for FastAPI) in a *thread* (Chapter 35), calls it with `httpx`, then stops it.

```
# Chapter 36 - Start a real server (uvicorn) in a thread, call it over the network,
then stop it
import socket
import threading
import time

import httpx
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/ping")
def ping():
    return {"alive": True}

with socket.socket() as s:
    # ask the system for a free port number
    s.bind(("127.0.0.1", 0))
    port = s.getsockname()[1]

server = uvicorn.Server(uvicorn.Config(app, host="127.0.0.1", port=port,
log_level="warning"))
thread = threading.Thread(target=server.run, daemon=True)
thread.start()
while not server.started:
    # wait until it is ready
    time.sleep(0.05)

r = httpx.get(f"http://127.0.0.1:{port}/ping") # a real HTTP call over the local
network
print(r.status_code, r.json())

server.should_exit = True
thread.join(timeout=5)
print("server still alive:", thread.is_alive())

200 {'alive': True}
server still alive: False
```

`server.should_exit = True` asks the server to stop politely; `thread.join()` waits until it is done. The final `False` proves the server is gone. The port is requested from the system (the number 0 means “choose a free one”) so it does not clash with other programs.

⚠ WARNING

A development server (uvicorn without protection, or Flask’s `app.run()`) is **not** meant to be opened to the internet. Serving an application safely (HTTPS, many processes, access limits) is covered in the Advanced level.

Running Project I5: a farm unit data API

The start of the web version of the “Information System for Forest, Orchard, Farming, Fisheries, Live-stock”: a list of **farm units** (plots, ponds, pens) that can be viewed, filtered by sector, and added to. Plot areas come from the Chapter 38 numbers (made-up data).

```
# Chapter 36 - Running project I5: API for farm unit data (first version)
from typing import Literal

from fastapi import FastAPI, HTTPException
from fastapi.testclient import TestClient
from pydantic import BaseModel, Field

class Unit(BaseModel):
    sector: Literal["forest", "orchard", "ricefield", "pond", "pen"]
    code: str
    area_ha: float = Field(gt=0)

app = FastAPI(title="Farm Information System")
unit_db: dict[str, Unit] = {
    "H1": Unit(sector="forest", code="H1", area_ha=9.0),
    "S1": Unit(sector="ricefield", code="S1", area_ha=3.0),
    "S2": Unit(sector="ricefield", code="S2", area_ha=2.25),
}

@app.get("/units", response_model=list[Unit])
def list_units(sector: str | None = None):
    return [u for u in unit_db.values() if sector is None or u.sector == sector]

@app.get("/units/{code}", response_model=Unit)
def one(code: str):
    if code not in unit_db:
        raise HTTPException(404, f"unit {code} does not exist")
    return unit_db[code]

@app.post("/units", response_model=Unit, status_code=201)
def add(unit: Unit):
    if unit.code in unit_db:
        raise HTTPException(409, f"unit {unit.code} already exists") # 409
    # conflict
    unit_db[unit.code] = unit
    return unit

c = TestClient(app)
print("ricefield:", [u["code"] for u in c.get("/units", params={"sector": "ricefield"}).json()])
print("add K1:", c.post("/units", json={"sector": "orchard", "code": "K1", "area_ha": 4.0}).status_code)
print("add K1 again:", c.post("/units", json={"sector": "orchard", "code": "K1", "area_ha": 4.0}).status_code)
print("wrong sector:", c.post("/units", json={"sector": "swamp", "code": "E1", "area_ha": 1}).status_code)
print("total area (ha):", sum(u["area_ha"] for u in c.get("/units").json()))
```

```
ricefield: ['S1', 'S2']
add K1: 201
add K1 again: 409
wrong sector: 422
total area (ha): 18.25
```

`Literal[...]` limits the sector to five choices, so the sector `swamp` is rejected with **422**. A code that already exists is rejected with **409**. The total area of 18.25 ha is $9.0 + 3.0 + 2.25$ (the initial units) + 4.0 (K1, which was added successfully). The data is still in memory; in later chapters this API will read the map data and the dashboard.

Exercises

1. In `01_flask_basics.py`, add an address `/harvest/<block>` that returns `{"block": ..., "yield_kg": 245.5}`. Test it with `test_client`.
2. In `03_fastapi_basics.py`, change `min_count: int` to `float` and try `?min_count=2.5`. Then change it back to `int`: what happens?
3. In `08_project_unit_api.py`, add a `DELETE /units/{code}` address that answers 204, and 404 if the code does not exist. Test both cases.
4. **Challenge:** add a `GET /summary` address that returns the total area per sector as a dict, using `defaultdict` or `dict.get`.

Chapter Summary

- The web = a **request** (method + address) and a **response** (status code + content, often JSON). **REST** arranges addresses as things and methods as actions.
- **Flask** is light and free-form; validation must be written by hand.
- **FastAPI** checks data types, uses Pydantic, filters answers with `response_model`, and writes the `/docs` documentation automatically.
- **Django** is all-in-one, good for complete sites.
- Apps can be tested without a server using test clients (`test_client`, `TestClient`, `Client`).
- A development server is not for opening to the internet.

Next, in Chapter 37: Python running inside the browser, with no server at all.

Chapter 37 — Python in the Browser: No Server, No Installing

Series I5 — A First Application: Web, Maps, and Dashboards

In this chapter, you will:

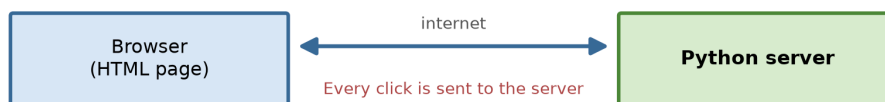
- Understand how Python can run **inside the browser** through **WebAssembly**
- Run Python code in **Pyodide** and understand its limits
- Build an interactive **PyScript** page without a server
- Meet **JupyterLite** (notebooks that run in the browser)
- Judge when this approach fits, and when it does not

All scripts and demo pages for this chapter are in the `code/ch37/` folder. On the web version, you can also [download them all at once \(zip\)](#).

In Chapter 36, every calculation happened on the **server**: the browser asks, the server calculates, the browser displays. Imagine an extension officer in a village with patchy signal who needs a **pond biomass calculator**. Starting and maintaining a server for a formula this small feels excessive. What if Python were **downloaded together with the page**, and all the calculation happened on the user's phone or laptop? That is Python in the browser.

Python on the server or Python in the browser?

The usual way



Pyodide / PyScript



Figure 0.1: Left to right: the usual way (calculation on the server) and Python in the browser (calculation on the user's computer).

WebAssembly, in brief

A browser natively understands only **JavaScript**. But modern browsers can also run **WebAssembly** (WASM for short): small, safe, fast machine code. Developers compile the Python interpreter (CPython) into WebAssembly, and the browser runs it like any other program inside a safe “sandbox”. The results:

Name	What it is
Pyodide	CPython compiled to WebAssembly, with many science libraries (NumPy and friends)

Name	What it is
PyScript	A framework on top of Pyodide (or MicroPython): write Python right in the HTML page
JupyterLite	JupyterLab/notebooks running entirely in the browser, using the Pyodide kernel

Pyodide: Python inside WebAssembly

Pyodide can run in the browser and also in Node.js. We use Node.js so we can **capture real output** for this book. The Python script below is **run by Pyodide**, not by the Python on your computer:

```
# Chapter 37 - Python running inside Pyodide
import platform
import sys

print("system:", sys.platform)          # 'emscripten' = the WebAssembly world
print("implementation:", platform.python_implementation(), "| Python",
      sys.version_info[0], sys.version_info[1])

def biomass_kg(fish_count, avg_weight_g):
    return fish_count * avg_weight_g / 1000

print("biomass of pond A (kg):", biomass_kg(800, 540.5))

# Calling JavaScript from Python
from js import Math

print("square root of 1600 via JavaScript:", Math.sqrt(1600))

# NumPy is available as a Pyodide package (loaded with loadPackage on the JavaScript
side)
import numpy as np

weights = np.array([480, 520, 505, 495, 540.5])
print("mean weight:", weights.mean(), "| ready to harvest (>=500 g):", int((weights
>= 500).sum()))

# A limit: no other processes and no ordinary network sockets
import subprocess

try:
    subprocess.run(["ls"])
except Exception as error:
    print("subprocess ->", type(error).__name__)
```

```
system: emscripten
implementation: CPython | Python 3 14
biomass of pond A (kg): 432.4
square root of 1600 via JavaScript: 40
mean weight: 508.1 | ready to harvest (>=500 g): 3
subprocess -> OSError
```

This output was captured for real, once, through the small wrapper `kode/ch37/node/run.mjs` (the `pyodide` library version 314.0.7 was installed with `npm install`, and NumPy was downloaded from Pyodide's official CDN). That is why the script is marked `@static`: the book build does not run it again.

Walk through the output:

- **system:** `emscripten` shows that Python runs in the WebAssembly world, not Windows or Linux.
- The Python is **CPython 3.14**, the same language as this book. The `biomass_kg` formula and NumPy work as usual (508.1 is the mean of the five weights).
- `from js import Math` calls **JavaScript** from Python: the two worlds can talk to each other.
- `subprocess` fails with `OSError`: the browser does not let a program start other processes. This is one **limit** of the sandbox.

i REMEMBER

For Pyodide, the version number follows the Python version: the **314** series means Python 3.14. We pin the version in the address (`.../pyodide/v314.0.7/full/pyodide.js`) so the page does not suddenly change behaviour when Pyodide is updated.

Pyodide straight in an HTML page

The page `code/ch37/demo/plot_map_pyodide.html` loads Pyodide from a CDN (a network that delivers files), runs Python that draws a **simple plot map** as SVG (vector graphics for the web), and pastes it into the page. Python calculates the areas with the shoelace formula; there is no server and no GeoPandas. Here is the skeleton:

```
<script src="https://cdn.jsdelivr.net/pyodide/v314.0.7/full/pyodide.js"></script>
<script>
    const pyodide = await loadPyodide();           // download and prepare Python
    const result = pyodide.runPython("1 + 2");     // run Python code
    await pyodide.loadPackage("numpy");           // Pyodide packages are loaded
when needed
</script>
```

I tried this demo in a real browser (the map and area table appeared, with H1 9.00 ha, K1 4.00 ha, S1 3.00 ha, S2 2.25 ha). To try it yourself, open `demo/plot_map_pyodide.html` through a simple local server (`python -m http.server` in the demo folder), because some browsers restrict pages opened directly from a file.

PyScript: Python inside HTML

With **PyScript**, Python is written right in the page using `<script type="py">`. The pond biomass calculator in `demo/calculator_pyscript.html` needs only two loading lines:

```
<link rel="stylesheet" href="https://pyscript.net/releases/2026.7.3/core.css">
<script type="module" src="https://pyscript.net/releases/2026.7.3/core.js"></script>
```

and then the Python:

```
from pyscript import document, when

@when("click", "#calc")
def calculate(event):
    count = float(document.querySelector("#count").value)
    weight = float(document.querySelector("#weight").value)
    document.querySelector("#result").innerHTML = f"Biomass: {count * weight / 1000:.1f} kg"
```

`@when("click", "#calc")` means: “when the button with id `calc` is clicked, run this function”. I tried that page in a real browser: clicking **Calculate** with the default entries (800 fish, 540.5 g) shows *Biomass: 432.4 kg*, the same as the calculations in Chapters 6 and 10. The `type="mpy"` option uses **MicroPython** (a mini Python, smaller download and faster start) instead of Pyodide; in this chapter we only tried `type="py"` (Pyodide).

Script 02 checks both demo pages: all their CDN addresses carry a **version number**.

```
# Chapter 37 - Checking the demo pages: CDN versions are pinned, no server needed
import re
from pathlib import Path

demo = Path(__file__).parent / "demo"
for page in sorted(demo.glob("*.html")):
    html = page.read_text(encoding="utf-8")
    addresses = sorted(set(re.findall(r'(?src|href)="(https://[^\"]+)"', html)))
    print(page.name)
    for a in addresses:
        print("  loads:", a)
        print("  versions pinned (a version number in the address):", all(re.search(r"\d+\.\d+\.\d+", a) for a in addresses))
```

```
calculator_pyscript.html
  loads: https://pyscript.net/releases/2026.7.3/core.css
  loads: https://pyscript.net/releases/2026.7.3/core.js
  versions pinned (a version number in the address): True
plot_map_pyodide.html
  loads: https://cdn.jsdelivr.net/pyodide/v314.0.7/full/pyodide.js
  versions pinned (a version number in the address): True
```

JupyterLite

JupyterLite is a Jupyter distribution that runs entirely in the browser with a Pyodide kernel. It is built with two packages: `jupyterlite-core` (the `jupyter lite build` command) and `jupyterlite-pyodide-kernel` (the Python kernel). The build result is a **static site** that can sit on ordinary file hosting, so it suits **teaching**: students just open a link, with nothing to install. According to the official documentation, the server that serves it must send `.wasm` files with the type `application/wasm`.

TECHNICAL STUFF (feel free to skip)

I did **not run** this part: `jupyter lite build` downloads many kernel files and I did not build it on the author's computer. The package versions listed on PyPI when this chapter was written: `jupyterlite-core` and `jupyterlite-pyodide-kernel` are both 0.8.6. The usage above comes from the JupyterLite documentation (the *standalone quickstart* page).

When it fits, when it does not

Fits	Fits poorly
Calculators and small helper tools	Big data (hundreds of MB)
Demos and teaching material with no installing	Heavy computing that needs a GPU
Tools that must work without signal once downloaded	Programs that need other processes, network sockets, or system files
Prototypes that need no server	Secrets: all code and data are downloaded by the user

Two important notes. First, **start-up time**: the core Pyodide files are some ten-odd megabytes; in my demo the page waited a few seconds before Python was ready. Second, **no secrets**: anything you put in the page (API keys, passwords, personal data) can be read by anyone who opens it.

⚠ WARNING

Not every library is available in Pyodide. Pure-Python libraries can usually be installed, but libraries with C code must already be compiled for WebAssembly. Check the Pyodide package list before planning an app.

Running Project I5: a plot area summary shown in the browser

A sensible pattern: **prepare the data on the author's computer** with GeoPandas (heavy), then **show the result in the browser** with PyScript (light). The script below calculates plot areas, then writes `main.py` (with the data embedded) and `index.html` to `_tmp/web/`:

```
# Chapter 37 - Running project I5: GeoPandas prepares the data, the browser shows it
with PyScript
import json
from pathlib import Path

import geopandas as gpd

web = Path(__file__).parent / "_tmp" / "web"
web.mkdir(parents=True, exist_ok=True)

# Step 1 (on the author's computer, ordinary Python): compute plot areas with GeoPandas
plots = gpd.read_file("data/plot.geojson").to_crs(32748)
plots["area_ha"] = (plots.area / 10_000).round(2)
data = plots[["code", "sector", "area_ha"]].to_dict("records")

# Step 2: write the program that will run in the browser (main.py) and its page
(web / "main.py").write_text(f'''from pyscript import document

DATA = {json.dumps(data)}

total = {{}}
for row in DATA:
    total[row["sector"]] = total.get(row["sector"], 0) + row["area_ha"]

text = "\\n".join(f"{{s:9s}} {{ha:6.2f}} ha" for s, ha in sorted(total.items()))
document.querySelector("#result").innerText = text
''', encoding="utf-8")

(web / "index.html").write_text("""<!doctype html>
<html lang="en"><head><meta charset="utf-8"><title>Area per sector</title>
<link rel="stylesheet" href="https://pyscript.net/releases/2026.7.3/core.css">
<script type="module" src="https://pyscript.net/releases/2026.7.3/core.js"></script></head>
<body><h1>Area per sector</h1><pre id="result">calculating...</pre>
<script type="py" src="./main.py"></script></body></html>
""", encoding="utf-8")

for file in sorted(web.iterdir()):
    print(file.name, "-", file.stat().st_size, "bytes")
print("\\ncontents of main.py (first 3 lines):")
print("\\n".join((web / "main.py").read_text(encoding="utf-8").splitlines()[:3])
[:140])
```

```
index.html - 394 bytes
main.py - 600 bytes
```



```
contents of main.py (first 3 lines):  
from pyscript import document
```

```
DATA = [{"code": "H1", "sector": "forest", "area_ha": 9.0}, {"code": "H2", "sector":  
"forest", "area_ha": 6.9}
```

I opened the resulting page through a local server and a browser: it showed `forest 15.93 ha`, `orchard 7.00 ha`, `ricefield 5.25 ha`, the same as the GeoPandas calculation in Chapter 38. Notice that `main.py` uses only plain Python: it does not need GeoPandas.

Exercises

1. Open `demo/calculator_pyscript.html` through a local server, change the fish count to 1250, and check the result against a hand calculation.
2. Change `calculator_pyscript.html` so it also shows an “estimated harvest (kg)” if 85 percent of the fish survive (an example figure).
3. Add one more plot to the `plots` dict in `plot_map_pyodide.html` and watch the map change.
4. **Challenge:** build a PyScript page that calculates the average weight from several numbers the user types, separated by commas.

Chapter Summary

- **WebAssembly** lets Python run inside the browser, in a safe sandbox.
- **Pyodide** = CPython on WebAssembly; **PyScript** = Python inside HTML; **JupyterLite** = notebooks in the browser.
- It fits demos, teaching, and small tools without a server; it does not fit big data, secrets, or programs that need other processes.
- Pin version numbers in CDN addresses. Code and data **are downloaded** by the user.

Next, in Chapter 38: your first map data: points, lines, and polygons with GeoPandas.

Chapter 38 — GIS 1: Vector Data: Maps You Can Calculate With

Series I5 — A First Application: Web, Maps, and Dashboards

In this chapter, you will:

- Understand spatial data: **points, lines, and polygons**
- Use **Shapely** (shapes) and **GeoPandas** (tables with shapes)
- Tell **GeoJSON**, **GeoPackage**, and **Shapefile** apart
- Understand the **CRS** (coordinate reference system) and **projections**, and avoid the degrees trap
- Calculate **plot areas**, **pond-to-river distances**, **buffers**, **overlays**, and **spatial joins**

All scripts for this chapter are in the `code/ch38/` folder. On the web version, you can also [download them all at once \(zip\)](#).

So far, “location” in this book has been just a name: pond A, plot H1. Yet field questions often concern **place**: *How many hectares of ricefield do we have? Which pond is closest to the river? Which plots fall inside the buffer zone?* **GIS** (Geographic Information System) answers them by storing the shape and position of things on earth as data, then calculating with it. This chapter covers **vector** data, things drawn with coordinates. (**Raster** data, grids of pixels like satellite photos, is in Chapter 39.)

All the data in this chapter is **made up** by scripts and the place is **fictional**, so the numbers are only for learning. It holds six plots (forest, orchard, ricefield), four ponds, three pens, one river, and two hamlets, stored in `code/ch38/data/` as GeoJSON files.

The three basic shapes of vector data

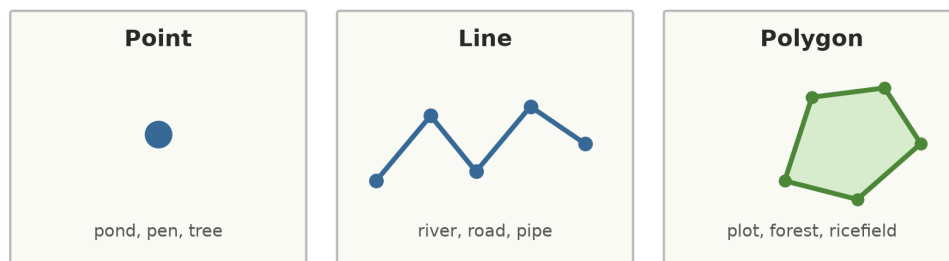


Figure 0.1: The three basic shapes of vector data.

Points, lines, polygons: Shapely

Shapely is a library for creating and calculating geometric shapes. It knows nothing about the earth: it only sees numbers on a flat plane.

```
# Chapter 38 - Points, lines, and polygons with Shapely
from shapely.geometry import LineString, Point, Polygon

# The coordinates below are in metres (a flat plane, same unit on both axes)
pond = Point(120, 285) # point: one location
river = LineString([(0, 330), (200, 310), (400, 290)]) # line: a sequence of points
plot = Polygon([(50, 100), (250, 100), (250, 250), (50, 250)]) # polygon: a closed area
```

```

print(pond.geom_type, river.geom_type, plot.geom_type)
print("river length (m):", round(river.length, 1))
print("plot area (m2)  :", plot.area)
print("plot perimeter (m):", plot.length)
print("plot centre point:", plot.centroid)

print("pond inside the plot? ", plot.contains(pond))
print("river crosses the plot? ", river.intersects(plot))
print("pond-to-river distance (m):", round(pond.distance(river), 1))
print("bounding box (minx, miny, maxx, maxy):", plot.bounds)

```

```

Point LineString Polygon
river length (m): 402.0
plot area (m2)  : 30000.0
plot perimeter (m): 700.0
plot centre point: POINT (150 175)
pond inside the plot? False
river crosses the plot? False
pond-to-river distance (m): 32.8
bounding box (minx, miny, maxx, maxy): (50.0, 100.0, 250.0, 250.0)

```

- **Point** for **one location** (pond, pen, tree), **LineString** for a **sequence of points** (river, road), **Polygon** for a **closed area** (plot).
- The plot area is 30000 m² (200 m times 150 m) and its perimeter 700 m. The river (from three points) is 402 m long.
- **contains**, **intersects**, **distance** answer questions of place: the pond is not inside the plot, the river does not cross the plot, and the pond-to-river distance is 32.8 m.

i REMEMBER

Shapely calculates in the **unit of the coordinate numbers**. If the coordinates are in metres, areas are in square metres. If they are in degrees, the “area” is in square degrees, which is useless. That is a CRS matter, covered shortly.

GeoPandas: a table that has shapes

GeoPandas adds one special column, **geometry**, to a pandas DataFrame (Chapter 23). Everything you know about pandas (filtering, **groupby**, merging) still applies. The engine that reads and writes map files behind the scenes is **pyogrio** (a replacement for fiona, which was not yet available for Python 3.14 when this chapter was written).

```

# Chapter 38 - GeoDataFrame and three file formats: GeoJSON, GeoPackage, Shapefile
import warnings
from pathlib import Path

import geopandas as gpd
from shapely.geometry import Point

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

# GeoDataFrame = a pandas table + one column named geometry + a CRS
pens = gpd.GeoDataFrame(
    {"code": ["D1", "D2"], "kind": ["cattle", "chicken"], "head_count_today_total":
    [12, 200]},
    geometry=[Point(106.8168, -6.9955), Point(106.8118, -6.9945)], # (longitude,
    latitude)
    crs="EPSG:4326",

```

```

)
print(pens)
print("CRS:", pens.crs)

pens.to_file(tmp / "pens.geojson", driver="GeoJSON")
pens.to_file(tmp / "pens.gpkg", layer="pens", driver="GPKG")
with warnings.catch_warnings(record=True) as w:
    warnings.simplefilter("always")
    pens.to_file(tmp / "pens.shp") # Shapefile
    print("Shapefile warning:", str(w[0].message))

print("Shapefile files:", sorted(p.name for p in tmp.glob("pens.*") if p.suffix not in
    (".geojson", ".gpkg")))
print("columns from Shapefile :", list(gpd.read_file(tmp / "pens.shp").columns))
print("columns from GeoPackage :", list(gpd.read_file(tmp / "pens.gpkg").columns))

```

```

code      kind  head_count_today_total      geometry
0   D1  cattle              12 POINT (106.8168 -6.9955)
1   D2  chicken            200 POINT (106.8118 -6.9945)
CRS: EPSG:4326
Shapefile warning: Column names longer than 10 characters will be truncated when saved
to ESRI Shapefile.
Shapefile files: ['pens.cpg', 'pens.dbf', 'pens.prj', 'pens.shp', 'pens.shx']
columns from Shapefile : ['code', 'kind', 'head_count', 'geometry']
columns from GeoPackage : ['code', 'kind', 'head_count_today_total', 'geometry']

```

The three formats you meet most often:

Format	Files	Strengths	Weaknesses
GeoJSON	one .geojson (text)	Easy for people and the web to read; generally EPSG:4326	Wasteful of space; slow for big data
GeoPackage	one .gpkg (SQLite)	One file, many layers , free column names, open standard	Not human-readable
Shapefile	five or more files (.shp, .shx, .dbf, .prj, .cpg)	Read by almost all older software	Column names limited to 10 letters

The output shows the Shapefile limit: the column `head_count_today_total` was **cut** to `head_count` (and you got a warning), while GeoPackage stored it whole. For new data, choose **GeoPackage**.

CRS: degrees or metres?

The earth is round, the map is flat. A **CRS** (*Coordinate Reference System*) explains what coordinate numbers mean. The two most important kinds:

- **EPSG:4326** (WGS 84): coordinates as **longitude and latitude in degrees**. Used by GPS and the web.
- **UTM projections**, for example **EPSG:32748** (UTM zone 48 South): coordinates **in metres** on a flat plane. Good for areas and distances within one region. Indonesia is divided into several UTM zones.

```

# Chapter 38 - Coordinate reference systems (CRS): degrees or metres?
import geopandas as gpd
from pyproj import CRS, Transformer

plots = gpd.read_file("data/plot.geojson")
print("file CRS:", plots.crs)
print("axis units:", [s.unit_name for s in plots.crs.axis_info])
print("corner of plot H1 (lon, lat):", [round(v, 5) for v in

```

```

plots.geometry.iloc[0].exterior.coords[0]])
print("'area' of H1 in square degrees:", plots.geometry.iloc[0].area) # meaningless!

utm = plots.estimate_utm_crs() # guesses the UTM zone from where
the data lies
print("matching UTM zone:", utm.to_string(), "|", utm.name)
flat = plots.to_crs(utm)
print("UTM axis units:", [s.unit_name for s in flat.crs.axis_info])
print("area of H1 (m2):", round(flat.geometry.iloc[0].area))

# Pyproj: move one point from 4326 to UTM. always_xy=True = order (longitude, latitude).
move = Transformer.from_crs(CRS.from_epsg(4326), CRS.from_epsg(32748), always_xy=True)
east, north = move.transform(106.8125, -6.9950)
print("point in UTM 48S: easting =", round(east, 1), "| northing =", round(north, 1))

```

```

file CRS: EPSG:4326
axis units: ['degree', 'degree']
corner of plot H1 (lon, lat): [106.81361, -6.99516]
'area' of H1 in square degrees: 7.363732839988365e-06
matching UTM zone: EPSG:32748 | WGS 84 / UTM zone 48S
UTM axis units: ['metre', 'metre']
area of H1 (m2): 90001
point in UTM 48S: easting = 700227.2 | northing = 9226418.1

```

- Our file comes in EPSG:4326, with axes in **degrees**. The “area” of H1 there is $7.36\text{e-}06$ square degrees: a number with no practical meaning.
- `estimate_utm_crs()` guesses the matching UTM zone from where the data lies: EPSG:32748.
- After `to_crs(...)`, the axes become **metres** and the area of H1 becomes 90001 m² (about 9 hectares; the 1 m² difference comes from rounding the GeoJSON coordinates to 7 decimals, about 1 cm).
- `pyproj` can move a single point by itself. `always_xy=True` makes sure the order is (longitude, latitude), because EPSG:4326 officially has the order (latitude, longitude), which often causes confusion.

⚠ WARNING

Never calculate areas or distances directly from data in degrees. Convert it first to a CRS with metre units using `to_crs`. This is the most common mistake in GIS. Every file carries its own CRS (`gdf.crs`): check it before combining data.

Plot areas and pond-to-river distances

```

# Chapter 38 - Plot area and pond-to-river distance
import geopandas as gpd

plots = gpd.read_file("data/plot.geojson").to_crs(32748) # metres
ponds = gpd.read_file("data/pond.geojson").to_crs(32748)
rivers = gpd.read_file("data/river.geojson").to_crs(32748)

plots["area_ha"] = (plots.area / 10_000).round(2)
print(plots[["code", "sector", "commodity", "area_ha"]].to_string(index=False))
print()
print("area per sector (ha):")
print(plots.groupby("sector")["area_ha"].sum().to_string())

river_line = rivers.geometry.iloc[0]
ponds["river_distance_m"] = ponds.distance(river_line).round(1)
print()
print(ponds[["code",

```

"kind",

```
"fish_count", "river_distance_m"]].to_string(index=False))
near = ponds[ponds["river_distance_m"] < 25]
print("ponds less than 25 m from the river:", list(near["code"]))
```

```
code    sector commodity  area_ha
H1     forest    Teak      9.00
H2     forest    Meranti   6.93
K1     orchard    coffee    4.00
K2     orchard    cocoa     3.00
S1     ricefield   rice     3.00
S2     ricefield   rice     2.25
```

```
area per sector (ha):
sector
forest      15.93
orchard      7.00
ricefield    5.25
```

```
code    kind  fish_count  river_distance_m
A  catfish      800           15.4
B  tilapia      500           29.8
C  catfish      650           20.0
D  tilapia      300          116.4
ponds less than 25 m from the river: ['A', 'C']
```

Forest is 15.93 ha (9.00 + 6.93), orchard 7.00 ha, ricefield 5.25 ha. `ponds.distance(line)` calculates the nearest distance from each pond to the river: A and C are under 25 m away, pond D is far (116.4 m). In a real case, a pond near a river is handy for water but at risk of flooding; the 25 m limit here is only an example.

Buffer and overlay

A **buffer** makes an area around a shape at a set distance. An **overlay** stacks two layers and calculates the pieces: intersection (the overlapping part) or difference (the non-overlapping part).

```
# Chapter 38 - Buffer (a protective zone) and overlay (stacking layers)
import geopandas as gpd

plots = gpd.read_file("data/plot.geojson").to_crs(32748)
rivers = gpd.read_file("data/river.geojson").to_crs(32748)

# Buffer: the area within 60 m of the river (60 is only a learning example)
zone = gpd.GeoDataFrame({"zone": ["buffer"]}, geometry=rivers.buffer(60),
crs=rivers.crs)
print("buffer zone area (ha):", round(zone.area.iloc[0] / 10_000, 2))

# Overlay intersection: the pieces of plots that fall inside the zone
inside = gpd.overlay(plots, zone, how="intersection")
inside["area_inside_m2"] = inside.area.round(0)
print("\nplots that enter the buffer zone:")
print(inside[["code", "sector", "area_inside_m2"]].to_string(index=False))

# Overlay difference: the part of each plot outside the zone
outside = gpd.overlay(plots, zone, how="difference")
outside["area_outside_ha"] = (outside.area / 10_000).round(2)
print("\nplot area outside the zone (ha):")
print(outside[["code", "area_outside_ha"]].to_string(index=False))
```

```
buffer zone area (ha): 13.24
```

```
plots that enter the buffer zone:
```

code	sector	area_inside_m2
K2	orchard	4.0
S1	ricefield	141.0
S2	ricefield	2333.0

```
plot area outside the zone (ha):
```

code	area_outside_ha
H1	9.00
H2	6.93
K1	4.00
K2	3.00
S1	2.99
S2	2.02

The 60 m zone on both sides of the river covers 13.24 ha. Three plots poke into it: S2 the most (2333 m²), S1 only 141 m², and K2 just 4 m². The last block shows what remains: S2 is left with 2.02 ha (from 2.25) and S1 with 2.99 ha. The 60 m figure is just an example; real river setback rules are set by local regulations.

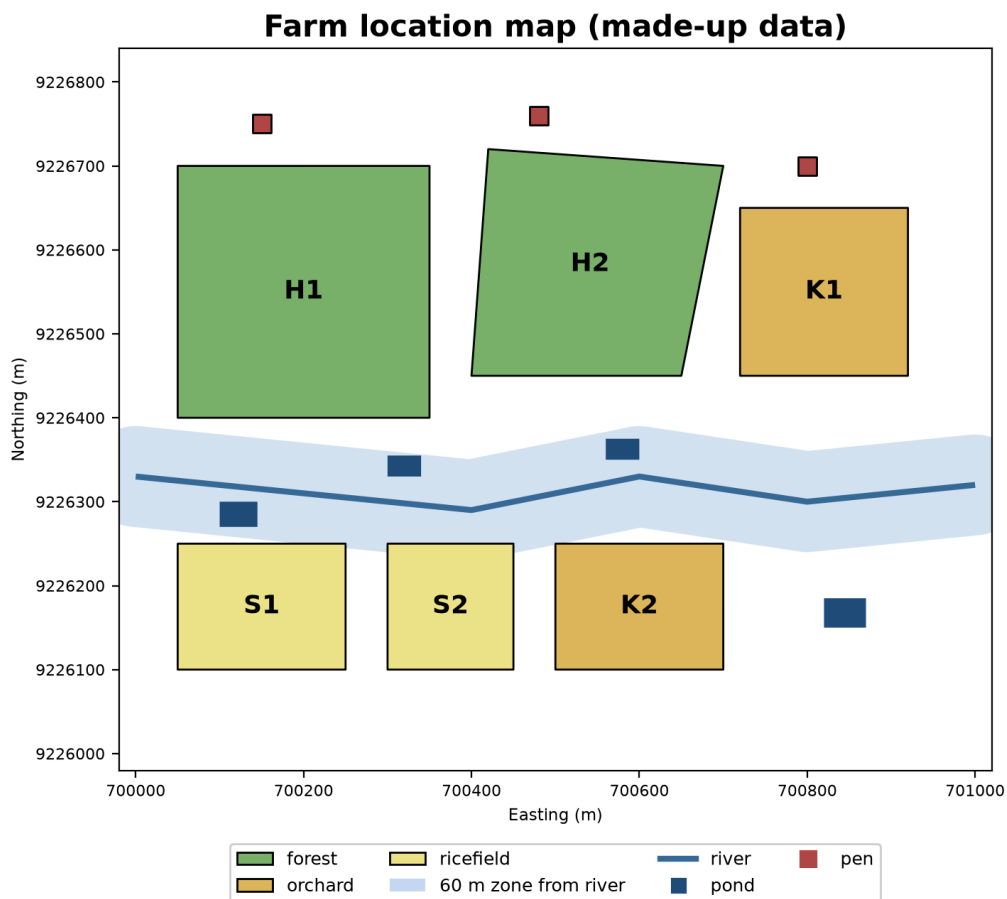


Figure 0.2: Farm location map (made-up data): plots, ponds, pens, the river, and the 60 m zone.

Spatial join: combining by location

In pandas, `merge` combines two tables by **matching column values**. In GeoPandas, a **spatial join** (`sjoin`) combines by **location relationship**: “which polygon is this point inside?”.

```
# Chapter 38 - Spatial join: combining tables by location
import warnings

import geopandas as gpd

hamlets = gpd.read_file("data/hamlet.geojson").to_crs(32748)
ponds = gpd.read_file("data/pond.geojson").to_crs(32748)
pens = gpd.read_file("data/pen.geojson").to_crs(32748)

# In which hamlet does the centre of each pond fall? (predicate="within": inside)
pond_points = ponds.copy()
pond_points["geometry"] = ponds.centroid
joined = gpd.sjoin(pond_points, hamlets[["name", "geometry"]], predicate="within")
print(joined[["code", "fish_count", "name"]].to_string(index=False))
print("\nfish per hamlet:")
print(joined.groupby("name")["fish_count"].sum().to_string())

# Nearest pen to each pond (sjoin_nearest) with the distance
nearest = gpd.sjoin_nearest(pond_points, pens[["code", "geometry"]],
                             distance_col="distance_m",
                             lsuffix="pond", rsuffix="pen")
print("\nnearest pen to each pond:")
print(nearest[["code_pond", "code_pen", "distance_m"]].round(0).to_string(index=False))

# Trap: different CRS -> only a warning, and the result is EMPTY
with warnings.catch_warnings(record=True) as w:
    warnings.simplefilter("always")
    wrong = gpd.sjoin(gpd.read_file("data/pond.geojson"), hamlets) # left 4326,
    right UTM
print("\ndifferent CRS ->", str(w[0].message).splitlines()[0])
print("number of result rows:", len(wrong))
```

```
code  fish_count  name
A      800  Upper
B      500  Upper
C      650  Lower
D      300  Lower
```

fish per hamlet:

```
name
Lower    950
Upper   1300
```

nearest pen to each pond:

```
code_pond code_pen  distance_m
A         D2        466.0
B         D2        442.0
C         D1        403.0
D         D1        534.0
```

different CRS -> CRS mismatch between the CRS of left geometries and the CRS of right geometries.

number of result rows: 0

- Ponds A and B fall in the Upper hamlet, C and D in the Lower, so fish per hamlet: Upper 1300, Lower 950.
- `sjoin_nearest` looks for **the nearest one**; pen D2 is nearest to ponds A and B, while D1 is nearest to C and D.
- **The trap at the end:** joining data with different CRSs raises no error, only a **warning**, and the result is **empty** (0 rows). Always `to_crs` first, and read warnings.

Installing GIS: pip or conda-forge?

On the author's computer, all GIS libraries installed with **pip** because they come as *wheels* (ready-made packages) that already carry the C libraries inside. That is not always so. The base libraries, **GDAL**, **GEOS**, and **PROJ**, are written in C/C++, and on some systems pip finds no matching wheel, so installation fails. **conda-forge** (a community package channel for **conda** or **mamba**) installs those C libraries **together with** the Python packages, so it is often smoother:

	pip	conda-forge
Command	<code>pip install geopandas rasterio</code>	<code>conda install -c conda-forge geopandas rasterio</code>
C libraries (GDAL, PROJ)	Inside the wheel, if one exists for your system	Installed as separate packages, with matching versions
Strength	Light, same way as other packages	Most reliable for GDAL, many other GIS tools
Weakness	Can fail if no wheel exists	Needs conda/mamba installed; environment separate from venv

TECHNICAL STUFF (feel free to skip)

I did **not run** the **conda** commands above: conda is not installed on the author's computer, so the conda-forge part has not been tested here. This comparison rests on how both tools generally work. If `pip install geopandas` works on your computer, you do not need conda.

Running Project I5: the farm location map

We bring it all together: read five layers, convert them to UTM, save them as **one GeoPackage with five layers**, summarise the area per sector, and make a quick map with `.plot()`. This GeoPackage is material for Chapter 40 (PostGIS and QGIS).

```
# Chapter 38 - Running project I5: the farm location map, saved to GeoPackage
from pathlib import Path

import geopandas as gpd
import matplotlib.pyplot as plt

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
layers = ["plot", "pond", "pen", "river", "hamlet"]
data = {name: gpd.read_file(f"data/{name}.geojson").to_crs(32748) for name in layers}

# One GeoPackage file holding many layers
gpkg = tmp / "farm_locations.gpkg"
gpkg.unlink(missing_ok=True)
for name, gdf in data.items():
    gdf.to_file(gpkg, layer=name, driver="GPKG")
print("layers inside the GeoPackage:", gpd.list_layers(gpkg)["name"].tolist())

# Summary per sector: plot area and number of points
plots = data["plot"]
print("\narea per sector (ha):")
area_ha = plots.assign(area_ha=plots.area / 10_000)
print(area_ha.groupby("sector")["area_ha"].sum().round(2).to_string())
print("number of ponds:", len(data["pond"]), "| number of pens:", len(data["pen"]))
```

```
# A quick map with .plot()
fig, ax = plt.subplots(figsize=(7, 5.6))
plots.plot(ax=ax, column="sector", legend=True, alpha=0.6, edgecolor="black")
data["river"].plot(ax=ax, color="tab:blue", linewidth=2)
data["pond"].plot(ax=ax, color="navy")
data["pen"].plot(ax=ax, color="red", markersize=40)
ax.set_title("Farm location map (made-up data)")
fig.savefig(tmp / "quick_map.png", dpi=100)
print("map saved:", (tmp / "quick_map.png").name)
```

```
layers inside the GeoPackage: ['plot', 'pond', 'pen', 'river', 'hamlet']
```

```
area per sector (ha):
sector
forest      15.93
orchard      7.00
ricefield    5.25
number of ponds: 4 | number of pens: 3
map saved: quick_map.png
```

`gpd.list_layers` proves that one file holds five layers. The quick map is saved in `_tmp/quick_map.png` (we make a tidier map in Chapter 40).

Exercises

1. In `01_shapely_geometry.py`, make a triangle polygon with points (0,0), (100,0), (0,60) and print its area. (Check it against half base times height.)
2. In `04_area_distance.py`, add a `river_distance_m` column for the **pens** and print the pen farthest from the river.
3. In `05_buffer_overlay.py`, change 60 m to 100 m. Which plots enter, and how much area enters in S2?
4. **Challenge:** for each hamlet, calculate the total plot area inside it using `overlay(plots, hamlets, how="intersection")` and `groupby`. Plot H2 crosses the hamlet border: check that its areas still add up to 6.93 ha.

Chapter Summary

- Vector data = **points, lines, polygons**. Shapely calculates the shapes; GeoPandas puts them in a table.
- Choose **GeoPackage** for new data; Shapefile cuts column names and needs many files.
- The **CRS** decides what coordinates mean. For areas and distances, convert to a CRS in metres (e.g. UTM) with `to_crs`.
- `area`, `distance`, `buffer`, `overlay`, `sjoin`, and `sjoin_nearest` answer field questions about place.
- A different CRS in `sjoin` or `overlay` gives only a warning and a wrong result: be careful.
- pip is enough when a wheel exists; conda-forge is more reliable for GDAL (not tested here).

Next, in Chapter 39: raster data: made-up satellite photos, pixels, and NDVI.

Chapter 39 — GIS 2: Raster Data: Aerial Photos You Can Calculate With

Series I5 — A First Application: Web, Maps, and Dashboards

In this chapter, you will:

- Understand **rasters**: pixels, resolution, and **bands**
- Read an image with **Rasterio** and move between pixels and coordinates
- Calculate **NDVI**, the plant greenness index
- **Clip** an image with a land boundary, then change its CRS and resolution
- Calculate **zonal statistics** per plot and pond, then save a **GeoTIFF**

All scripts for this chapter are in the `code/ch39/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Chapter 38 drew land with lines. But a satellite photo or drone photo contains no lines: it holds a **grid of tiny squares**, each with one number. That is a **raster**. Imagine a chequered cloth laid over your garden; each square is painted the average colour of the ground beneath it. A square is called a **pixel**.

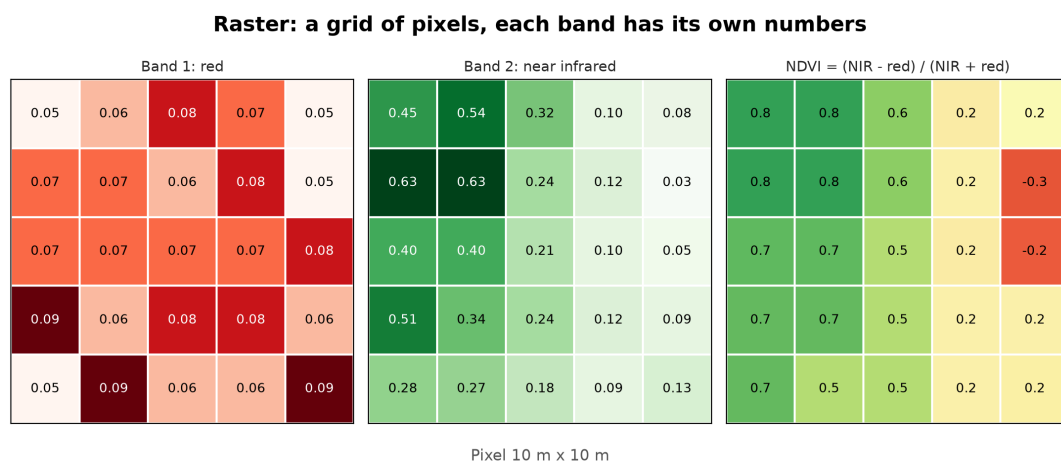


Figure 0.1: A raster = a grid of pixels. Each band has its own numbers; NDVI is calculated from two bands.

Three key terms:

- **Pixel**: one square of the grid, with one value per band.
- **Resolution**: the side length of one pixel on the ground. A 10 m resolution means one pixel represents 10 m by 10 m (100 m²). The smaller the number, the finer the image, and the bigger the file.
- **Band**: one “layer” of values. An ordinary photo has three bands (red, green, blue). Satellite images have other bands the eye cannot see, such as **near infrared (NIR)**.

i REMEMBER

Healthy plants **reflect a lot** of near infrared and **absorb a lot** of red light. Water hardly reflects any infrared. This difference is used to measure “greenness”.

Downloading big images for this book is not allowed, so we make a **made-up satellite image** with a script: a grid of 100 columns by 80 rows, 10 m pixels, covering the same 1 km × 800 m area as Chapter 38, with two bands (red and NIR) and a little random noise. Each plot is deliberately given a certain greenness.

```
# Chapter 39 - Making a made-up satellite image (2 bands: red and near infrared)
from pathlib import Path

import geopandas as gpd
import numpy as np
import rasterio
from rasterio.features import rasterize
from rasterio.transform import from_origin

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

# A grid of 100 columns x 80 rows, 10 m pixels, top-left corner at (700000, 9226800)
# UTM 48S
transform = from_origin(700000, 9226800, 10, 10)
shape = (80, 100)

plots = gpd.read_file("data/plot.geojson").to_crs(32748)
ponds = gpd.read_file("data/pond.geojson").to_crs(32748)
rivers = gpd.read_file("data/river.geojson").to_crs(32748)

# A "land cover map": each pixel gets the NDVI value we want it to have
target_ndvi = np.full(shape, 0.15) # bare soil
plot_value = {"H1": 0.80, "H2": 0.78, "K1": 0.60, "K2": 0.55, "S1": 0.45, "S2": 0.70}
for _, row in plots.iterrows():
    mask = rasterize([(row.geometry, 1)], out_shape=shape, transform=transform,
dtype="uint8")
    target_ndvi[mask == 1] = plot_value[row["code"]]
water = list(ponds.geometry) + list(rivers.buffer(8).geometry) # ponds and the
river body
target_ndvi[rasterize(water, out_shape=shape, transform=transform, dtype="uint8") ==
1] = -0.20

# From the target NDVI, build the red and NIR bands (plus a little random noise)
rng = np.random.default_rng(42)
target_ndvi = target_ndvi + rng.normal(0, 0.03, shape)
red = rng.uniform(0.05, 0.09, shape)
nir = red * (1 + target_ndvi) / (1 - target_ndvi)
image = np.stack([red, nir]).astype("float32")

profile = dict(driver="GTiff", height=80, width=100, count=2, dtype="float32",
crs="EPSG:32748", transform=transform)
with rasterio.open(tmp / "synthetic_image.tif", "w", **profile) as target:
    target.write(image)
    target.set_band_description(1, "red")
    target.set_band_description(2, "nir")
print("saved:", (tmp / "synthetic_image.tif").name, "| shape:", image.shape)

# Proof: identical to the file provided in data/
with rasterio.open("data/synthetic_image.tif") as existing:
    print("identical to data/synthetic_image.tif:", np.array_equal(existing.read(),
image))
```

```
saved: synthetic_image.tif | shape: (2, 80, 100)
identical to data/synthetic_image.tif: True
```

This script makes the image in `_tmp/` and **proves** it is identical to the provided file `data/synthetic_image.tif`. The following scripts read the provided file. Two new tools appear here:

`rasterize` turns polygons into pixels, and `from_origin(x, y, width, height)` sets the grid's **position** on earth (top-left corner and pixel size).

Reading rasters with Rasterio

Rasterio reads and writes rasters (using GDAL behind the scenes). Each band is read as a **NumPy array** (Chapter 23).

```
# Chapter 39 - Reading a raster: pixels, resolution, bands, and coordinates
import rasterio

with rasterio.open("data/synthetic_image.tif") as image:
    print("size (width x height):", image.width, "x", image.height, "pixels")
    print("number of bands:", image.count, "| descriptions:", image.descriptions)
    print("data type:", image.dtypes[0])
    print("CRS:", image.crs)
    print("resolution (m):", image.res)
    print("bounds (left, bottom, right, top):", tuple(image.bounds))

    red = image.read(1)          # band 1 -> a two-dimensional NumPy array
    nir = image.read(2)
    print("\narray shape:", red.shape, "(rows, columns)")
    print("red: min %.3f max %.3f" % (red.min(), red.max()))
    print("nir: min %.3f max %.3f" % (nir.min(), nir.max()))

    # Pixel (row 10, column 20) -> the map coordinates of its centre, and back
    x, y = image.xy(10, 20)
    print("\ncentre of pixel (10, 20):", x, y)
    print("that coordinate maps back to (row, column):", image.index(x, y))

    # The value at one point, e.g. inside pond A (easting 700122, northing 9226285)
    print("[red, nir] at pond A:", [round(float(v), 3) for v in
next(image.sample([(700122, 9226285)]))])
```

```
size (width x height): 100 x 80 pixels
number of bands: 2 | descriptions: ('merah', 'nir')
data type: float32
CRS: EPSG:32748
resolution (m): (10.0, 10.0)
bounds (left, bottom, right, top): (700000.0, 9226000.0, 701000.0, 9226800.0)

array shape: (80, 100) (rows, columns)
red: min 0.050 max 0.090
nir: min 0.030 max 1.314

centre of pixel (10, 20): 700205.0 9226695.0
that coordinate maps back to (row, column): (10, 20)
[red, nir] at pond A: [0.057, 0.038]
```

- The image is 100 × 80 pixels, with two bands named `red` and `nir`, type `float32`, CRS `EPSG:32748` (metres), resolution 10 m.
- `bounds` shows where it lies: from easting 700000 to 701000 and northing 9226000 to 9226800.
- The array has shape `(80, 100)`: **rows first, then columns**. That is NumPy's order, the opposite of the `(x, y)` we usually think of on a map.
- `xy(row, column)` turns a pixel number into the coordinates of its centre; `index(x, y)` reverses it. `sample` reads the value at one point: inside pond A, red is 0.057 and NIR 0.038. NIR is lower than red, a typical sign of water.

NDVI: measuring greenness

NDVI (*Normalized Difference Vegetation Index*) = $(\text{NIR} - \text{Red}) \div (\text{NIR} + \text{Red})$. The result is between -1 and 1: water is negative, bare soil is around 0.1 to 0.2, and dense plants approach 0.8 and above.

```
# Chapter 39 - NDVI: the plant greenness index
from pathlib import Path

import numpy as np
import rasterio

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

with rasterio.open("data/synthetic_image.tif") as image:
    red, nir = image.read(1), image.read(2)
    profile = image.profile

# NDVI = (NIR - Red) / (NIR + Red), a value between -1 and 1
ndvi = (nir - red) / (nir + red)
print("NDVI: min %.2f | mean %.2f | max %.2f" % (ndvi.min(), ndvi.mean(), ndvi.max()))

# Simple classes (the thresholds are only learning examples)
classes = np.digitize(ndvi, [0.0, 0.3, 0.6])      # 0=water, 1=bare, 2=medium, 3=dense
names = ["water", "bare soil", "medium vegetation", "dense vegetation"]
for i, n in enumerate(names):
    count = int((classes == i).sum())
    print(f"{n:18s}: {count:5d} pixels = {count * 100 / 10_000:5.1f} ha")

# Save the NDVI as a one-band GeoTIFF
profile.update(count=1, dtype="float32", nodata=-9999.0)
with rasterio.open(tmp / "ndvi.tif", "w", **profile) as target:
    target.write(ndvi.astype("float32"), 1)
print("saved:", "ndvi.tif")
```

```
NDVI: min -0.29 | mean 0.33 | max 0.89
water      :    216 pixels =   2.2 ha
bare soil  :   4964 pixels =  49.6 ha
medium vegetation :   777 pixels =   7.8 ha
dense vegetation :  2043 pixels =  20.4 ha
saved: ndvi.tif
```

The mean NDVI of the whole image is 0.33 and the smallest is -0.29 (water). The classes are made with `np.digitize`; the thresholds 0, 0.3, and 0.6 are only learning examples (real values depend on the sensor and crop). For instance, dense vegetation is $2043 \text{ pixels} \times 100 \text{ m}^2 = 20.4 \text{ ha}$. The operation $(\text{nir} - \text{red}) / (\text{nir} + \text{red})$ works on the **whole grid at once**, with no loop: the power of NumPy. The result is saved as a GeoTIFF (nodata marked -9999).

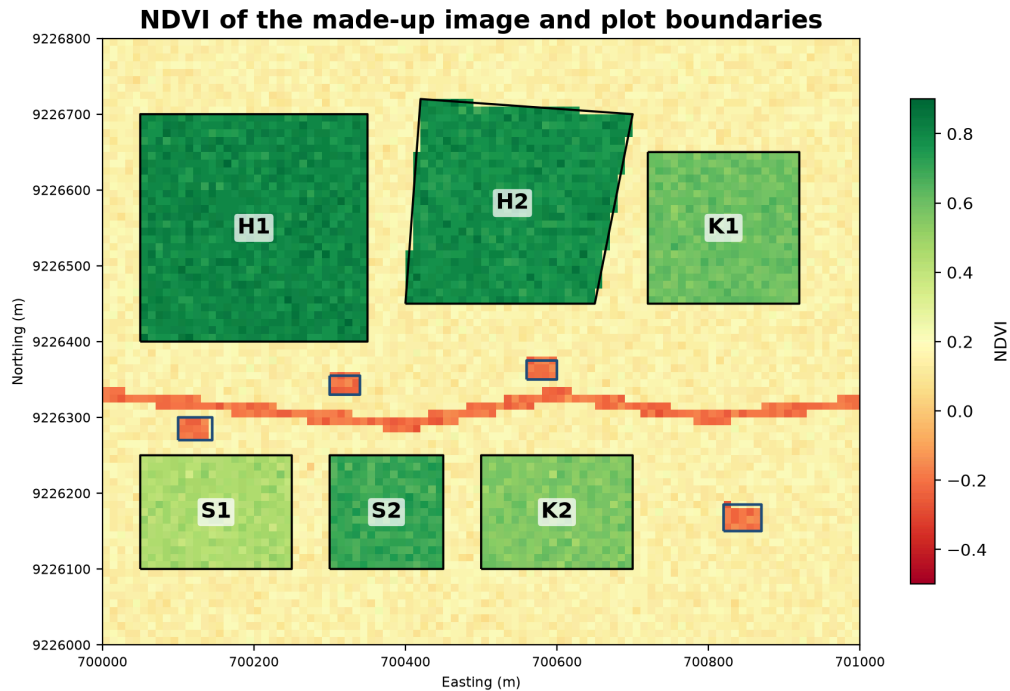


Figure 0.2: NDVI of the made-up image with plot and pond boundaries. Dark green = dense, orange = water.

Clipping an image with a land boundary

Usually we only need the image inside a certain piece of land. `rasterio.mask.mask` clips it with a polygon.

```
# Chapter 39 - Clipping the image with a land boundary
from pathlib import Path

import geopandas as gpd
import numpy as np
import rasterio
from rasterio.mask import mask

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

plots = gpd.read_file("data/plot.geojson")
with rasterio.open("data/synthetic_image.tif") as image:
    s1 = plots[plots["code"] == "S1"].to_crs(image.crs)      # vector CRS = raster CRS
    clipped, transform = mask(image, s1.geometry, crop=True, nodata=np.nan)
    profile = image.profile

    red, nir = clipped
    ndvi = (nir - red) / (nir + red)
    print("clip shape (band, rows, columns):", clipped.shape)
    print("pixels inside S1:", int(np.isfinite(ndvi).sum()))
    print("mean NDVI of S1: %.2f" % np.nanmean(ndvi))

    profile.update(height=clipped.shape[1], width=clipped.shape[2], transform=transform,
        nodata=np.nan)
    with rasterio.open(tmp / "clip_S1.tif", "w", **profile) as target:
        target.write(clipped)
    print("saved:", "clip_S1.tif")
```



```
clip shape (band, rows, columns): (2, 17, 21)
pixels inside S1: 300
mean NDVI of S1: 0.45
saved: clip_S1.tif
```

The clip is 17 rows by 21 columns: a box containing S1 (originally 15 rows × 20 columns of pixels), a bit bigger because the box edges are rounded outward (I did not check the exact cause). What really lies inside the plot: **300 pixels = 3 hectares**, equal to the area of S1 in Chapter 38. Its mean NDVI is 0.45. Notice `.to_crs(image.crs)`: the polygon **must** have the same CRS as the raster. If you forget:

```
# Chapter 39 - Trap: the land boundary has a different CRS from the raster
import geopandas as gpd
import rasterio
from rasterio.mask import mask

plots = gpd.read_file("data/plot.geojson")          # EPSG:4326 (degrees), forgot to_crs!
with rasterio.open("data/synthetic_image.tif") as image: # raster EPSG:32748 (metres)
    print("raster CRS:", image.crs, "| vector CRS:", plots.crs)
    mask(image, plots.geometry, crop=True)
```

```
raster CRS: EPSG:32748 | vector CRS: EPSG:4326
Traceback (most recent call last):
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\mask.py", line 80, in
raster_geometry_mask
    window = geometry_window(dataset, shapes, pad_x=pad_x, pad_y=pad_y)
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\features.py", line 522, in
geometry_window
    bounding_window = bounding_window.intersection(raster_window)
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\windows.py", line 787, in
intersection
    return intersection([self, other])
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\windows.py", line 125, in
wrapper
    return function(*args[0])
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\windows.py", line 239, in
intersection
    return functools.reduce(_intersection, windows)
    ~~~~~^~~~~~
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\windows.py", line 257, in
_intersection
    raise WindowError(f"Intersection is empty {w1} {w2}")
rasterio.errors.WindowError: Intersection is empty Window(col_off=-69990,
row_off=922680, width=1, height=1) Window(col_off=0, row_off=0, width=100, height=80)
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "08_wrong_crs_mask.py", line 9, in <module>
    mask(image, plots.geometry, crop=True)
    ~~~~^~~~~~
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\mask.py", line 180, in mask
    shape_mask, transform, window = raster_geometry_mask(
    ~~~~~^~~~~~
    dataset, shapes, all_touched=all_touched, invert=invert, crop=crop,
    ~~~~~^~~~~~
    pad=pad, pad_width=pad_width)
    ~~~~~^~~~~~
  File "<folder-proyek>\.venv\Lib\site-packages\rasterio\mask.py", line 86, in
raster_geometry_mask
```

```
raise ValueError('Input shapes do not overlap raster.')
ValueError: Input shapes do not overlap raster.
```

Rasterio refuses with `Input shapes do not overlap raster`: degree coordinates (around 106) do not overlap metre coordinates (around 700000). Luckily there is an error here. Even without an error, a result like this should always be suspected.

Reproject and resample

Reproject changes a raster's CRS; **resample** changes pixel size. Both force the computer to **recalculate** pixel values, and there is a choice of method: `bilinear` (a weighted average of neighbours, good for continuous data like NDVI), `average`, or `nearest` (copy the nearest neighbour, a must for class data such as a land-cover map, so no "class 1.5" appears).

```
# Chapter 39 - Reproject (change the CRS) and resample (change the resolution)
import numpy as np
import rasterio
from rasterio.enums import Resampling
from rasterio.warp import calculate_default_transform, reproject

with rasterio.open("data/synthetic_image.tif") as image:
    nir = image.read(2)

    # Reproject: UTM (metres) -> WGS 84 (degrees)
    transform, width, height = calculate_default_transform(
        image.crs, "EPSG:4326", image.width, image.height, *image.bounds)
    result = np.zeros((height, width), dtype="float32")
    reproject(source=nir, destination=result, src_transform=image.transform,
              src_crs=image.crs,
                                dst_transform=transform, dst_crs="EPSG:4326",
              resampling=Resampling.bilinear)
    print("before:", image.width, "x", image.height, "pixels, unit metres")
    print("after :", width, "x", height, "pixels, pixel size (degrees): %.6f" %
          transform.a)

    # Resample: read directly at 3x coarser resolution (30 m), values averaged
    coarse = image.read(2, out_shape=(image.height // 3, image.width // 3),
                          resampling=Resampling.average)
    print("\nresolution 10 m :", nir.shape, "pixels")
    print("resolution 30 m :", coarse.shape, "pixels")
    print("mean NIR 10 m vs 30 m: %.4f vs %.4f" % (nir.mean(), coarse.mean()))
```

```
before: 100 x 80 pixels, unit metres
after : 100 x 80 pixels, pixel size (degrees): 0.000090

resolution 10 m : (80, 100) pixels
resolution 30 m : (26, 33) pixels
mean NIR 10 m vs 30 m: 0.2249 vs 0.2249
```

After reprojecting to EPSG:4326, the pixel size becomes about 0.00009 degrees (roughly 10 m). Going from 10 m to 30 m resolution gives a grid of 26 × 33 pixels (a third of the size, rounded down). The mean NIR before and after is the same (0.2249): `Resampling.average` preserves the mean.

Zonal statistics

Zonal statistics summarise raster values inside each **zone** (polygon): mean, smallest, largest. We make a **mask** for each polygon with `geometry_mask`, then calculate with NumPy.

```
# Chapter 39 - Zonal statistics: summarising raster values inside each polygon
import geopandas as gpd
```

```

import rasterio
from rasterio.features import geometry_mask

with rasterio.open("data/synthetic_image.tif") as image:
    red, nir = image.read(1), image.read(2)
    ndvi = (nir - red) / (nir + red)
    plots = gpd.read_file("data/plot.geojson").to_crs(image.crs)
    ponds = gpd.read_file("data/pond.geojson").to_crs(image.crs)

    def zonal(zones):
        result = []
        for _, z in zones.iterrows():
            # invert=True: True INSIDE the polygon, False outside
            inside = geometry_mask([z.geometry], out_shape=ndvi.shape,
transform=image.transform, invert=True)
            values = ndvi[inside]
            result.append((z["code"], int(inside.sum()), values.mean(), values.min(),
values.max()))
        return result

    print("zone      pixels  NDVI_mean  NDVI_min  NDVI_max")
    for code, n, mean, low, high in zonal(plots):
        print(f"plot {code:>2s} {n:4d} {mean:9.2f} {low:8.2f} {high:9.2f}")
    for code, n, mean, low, high in zonal(ponds):
        print(f"pond {code:>2s} {n:4d} {mean:9.2f} {low:8.2f} {high:9.2f}")

```

	zone	pixels	NDVI_mean	NDVI_min	NDVI_max
plot	H1	900	0.80	0.71	0.89
plot	H2	695	0.78	0.67	0.88
plot	K1	400	0.60	0.51	0.68
plot	K2	300	0.55	0.45	0.63
plot	S1	300	0.45	0.36	0.55
plot	S2	225	0.70	0.62	0.76
pond	A	12	-0.20	-0.24	-0.15
pond	B	12	-0.21	-0.25	-0.15
pond	C	12	-0.20	-0.25	-0.14
pond	D	16	-0.19	-0.26	-0.14

Forest H1 and H2 are densely green (0.80 and 0.78), ricefield S1 is lowest (0.45; in this made-up data the value was deliberately set low, as if the rice were young), and all four ponds have negative NDVI (around -0.2): water. The pixel counts also match the areas: H1 900 pixels × 100 m² = 9 ha.

Running Project I5: NDVI per plot

The zonal result goes back into the plot table and gets flagged.

```

# Chapter 39 - Running project I5: NDVI per plot, saved to GeoPackage
from pathlib import Path

import geopandas as gpd
import numpy as np
import rasterio
from rasterio.features import geometry_mask

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)

with rasterio.open("data/synthetic_image.tif") as image:
    red, nir = image.read(1), image.read(2)
    ndvi = (nir - red) / (nir + red)

```

```

plots = gpd.read_file("data/plot.geojson").to_crs(image.crs)
mean = []
for geom in plots.geometry:
    # invert=True: True INSIDE the polygon
    inside = geometry_mask([geom], out_shape=ndvi.shape, transform=image.transform,
invert=True)
    mean.append(round(float(ndvi[inside].mean()), 2))

plots["ndvi_mean"] = mean
plots["status"] = np.where(plots["ndvi_mean"] < 0.5, "check it", "good")    # threshold
0.5 = example
plots["area_ha"] = (plots.area / 10_000).round(2)
print(plots[["code", "sector", "commodity", "area_ha", "ndvi_mean",
"status"]].to_string(index=False))

plots.to_file(tmp / "plots_ndvi.gpkg", layer="plots_ndvi", driver="GPKG")
print("\nsaved:   plots_ndvi.gpkg | columns:", list(gpd.read_file(tmp /
"plots_ndvi.gpkg").columns))

```

code	sector	commodity	area_ha	ndvi_mean	status
H1	forest	Teak	9.00	0.80	good
H2	forest	Meranti	6.93	0.78	good
K1	orchard	coffee	4.00	0.60	good
K2	orchard	cocoa	3.00	0.55	good
S1	ricefield	rice	3.00	0.45	check it
S2	ricefield	rice	2.25	0.70	good

```

saved: plots_ndvi.gpkg | columns: ['code', 'sector', 'commodity', 'ndvi_mean',
'status', 'area_ha', 'geometry']

```

Only S1 (0.45) has the status “check it” because its NDVI is below the 0.5 threshold, which here is just an example number. The table is saved to a GeoPackage: vector data and raster results finally come together. Remember that this image is **made up**; with a real image, NDVI should be compared with the planting season and field observation.

Exercises

1. In `03_ndvi.py`, change the class thresholds to `[0.0, 0.2, 0.5]` and watch the area of each class change.
2. In `04_clip_to_boundary.py`, clip plot K1 (the coffee orchard) and print its mean NDVI.
3. In `06_zonal_statistics.py`, add a standard deviation column (`values.std()`). Which plot is the most uniform?
4. **Challenge:** make a column `pixels_below_0_5` for each plot and show its percentage. Compare it with the “check it” status.

Chapter Summary

- A **raster** = a grid of pixels; **resolution** = pixel size on the ground; a **band** = a layer of values.
- **Rasterio** reads bands as NumPy arrays; `xy` and `index` move between pixels and coordinates.
- **NDVI** = $(\text{NIR} - \text{red}) / (\text{NIR} + \text{red})$: water is negative, bare soil low, dense plants high.
- `mask` clips with a polygon of the same CRS; `reproject` and resampling recalculate pixels (choose the method to suit the data type).
- **Zonal statistics** summarise a raster per polygon; the result can be joined to a vector table.

Next, in Chapter 40: static and interactive maps, PostGIS, and driving QGIS from Python.

Chapter 40 — GIS 3: Maps and Integration: PostGIS and QGIS

Series I5 — A First Application: Web, Maps, and Dashboards

In this chapter, you will:

- Make **static maps** (Matplotlib) and **interactive maps** (folium)
- Store and query spatial data in **PostGIS** from Python
- Drive **QGIS** with **PyQGIS** without opening its window
- Understand how PyQGIS differs in **QGIS 4** (Qt6) compared with QGIS 3
- Store farm location data in PostGIS **and** GeoPackage: the running project

All scripts for this chapter are in the `code/ch40/` folder. On the web version, you can also [download them all at once \(zip\)](#).

The two earlier GIS chapters let you **calculate**. Now we **present** and **store**. A map is how people read spatial data; a spatial database is a lasting warehouse; and QGIS is the “map kitchen” used by many organisations. Python can sit in the middle of all three.

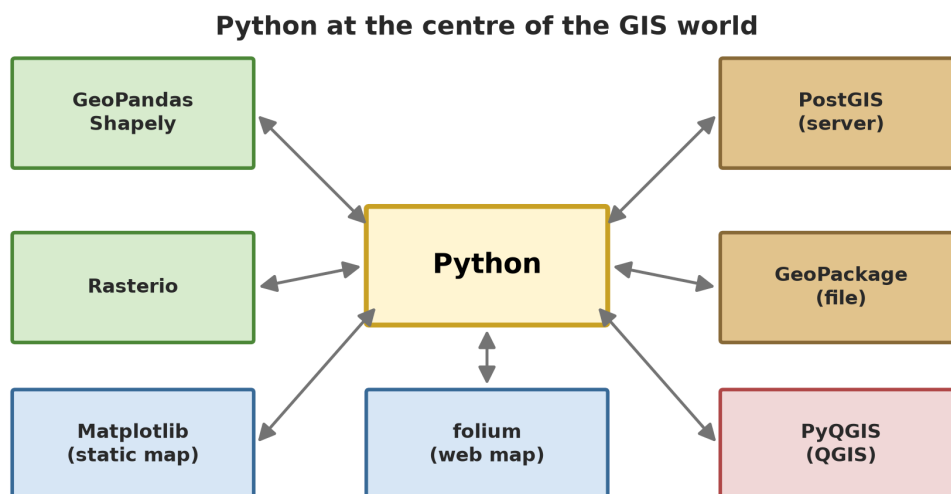


Figure 0.1: Python at the centre: reading and writing to many kinds of storage and map presenters.

Static maps with Matplotlib

A static map is an image (PNG, PDF) for reports and books. GeoPandas draws through Matplotlib (Chapter 25), so all of Matplotlib’s controls are available: titles, legends, labels.

```
# Chapter 40 - A static map with GeoPandas and Matplotlib
from pathlib import Path

import geopandas as gpd
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

tmp = Path(__file__).parent / "_tmp"
```

```

tmp.mkdir(exist_ok=True)
plots = gpd.read_file("data/plot.geojson").to_crs(32748)
ponds = gpd.read_file("data/pond.geojson").to_crs(32748)
pens = gpd.read_file("data/pen.geojson").to_crs(32748)
rivers = gpd.read_file("data/river.geojson").to_crs(32748)

colour = {"forest": "#4f8a3b", "orchard": "#d9a441", "ricefield": "#e8dc7a"}
fig, ax = plt.subplots(figsize=(7, 5.6), dpi=100)
for sector, group in plots.groupby("sector"):
    group.plot(ax=ax, color=colour[sector], edgecolor="black", linewidth=0.8)
rivers.plot(ax=ax, color="#3b6a9a", linewidth=2.5)
ponds.plot(ax=ax, color="#1f4e79")
pens.plot(ax=ax, color="#b04a4a", markersize=60, marker="s")
for _, p in plots.iterrows():
    ax.annotate(p["code"], (p.geometry.centroid.x, p.geometry.centroid.y), ha="center",
weight="bold")

ax.set_title("Farm location map (made-up data)")
ax.set_xlabel("Easting (m, UTM 48S)")
ax.set_ylabel("Northing (m, UTM 48S)")
ax.ticklabel_format(style="plain", useOffset=False)
ax.set_aspect("equal")
# 1 m on the X axis =
1 m on the Y axis
legend = [Line2D([0], [0], marker="s", color="w", markerfacecolor=c, markersize=12,
label=s)
    for s, c in colour.items()]
legend += [Line2D([0], [0], color="#3b6a9a", lw=2.5, label="river"),
    Line2D([0], [0], marker="s", color="w", markerfacecolor="#1f4e79",
markersize=9, label="pond"),
    Line2D([0], [0], marker="s", color="w", markerfacecolor="#b04a4a",
markersize=9, label="pen")]
ax.legend(handles=legend, loc="upper left", fontsize=8)
fig.tight_layout()
fig.savefig(tmp / "static_map.png")
print("map saved:", "static_map.png", "| size in pixels:", fig.get_size_inches() *
fig.dpi)

```

```
map saved: static_map.png | size in pixels: [700. 560.]
```

What sets a map apart from an ordinary chart:

- **ax.set_aspect("equal")**: 1 metre on the X axis is drawn as long as 1 metre on the Y axis. Without it, square plots look squashed.
- The data is drawn in **UTM** (metres), so the axes can be labelled in metres and distances read correctly.
- Colours are chosen per sector with **groupby**, then the legend is built by hand from **Line2D**.

That 700 × 560 pixel image is saved in `_tmp/static_map.png`. You already saw a map of the same data with the 60 m zone in Chapter 38.

Interactive maps with folium

folium wraps the JavaScript library **Leaflet** and produces an **HTML file** that can be dragged and zoomed in a browser, complete with markers and *popups*.

```

# Chapter 40 - An interactive map with folium (the result is an HTML file)
from pathlib import Path

import folium
import geopandas as gpd

tmp = Path(__file__).parent / "_tmp"

```

```

tmp.mkdir(exist_ok=True)
plots = gpd.read_file("data/plot.geojson")          # folium needs EPSG:4326 (degrees)
ponds = gpd.read_file("data/pond.geojson")
pens = gpd.read_file("data/pen.geojson")
rivers = gpd.read_file("data/river.geojson")

centre = plots.union_all().centroid
farm_map = folium.Map(location=[centre.y, centre.x], zoom_start=16,
tiles="OpenStreetMap") # order: latitude, longitude

folium.GeoJson(plots, name="plots",
                tooltip=folium.GeoJsonTooltip(fields=["code", "sector",
"commodity"])).add_to(farm_map)
folium.GeoJson(rivers, name="river", style_function=lambda f: {"color": "#3b6a9a",
"weight": 4}).add_to(farm_map)
for _, k in ponds.iterrows():
    folium.Marker([k.geometry.centroid.y, k.geometry.centroid.x],
                  popup=f"Pond {k['code']}: {k['fish_count']} {k['kind']}",
                  icon=folium.Icon(color="blue", icon="tint")).add_to(farm_map)
for _, k in pens.iterrows():
    folium.Marker([k.geometry.y, k.geometry.x], popup=f"Pen {k['code']}: {k['kind']}",
                  icon=folium.Icon(color="red")).add_to(farm_map)
folium.LayerControl().add_to(farm_map)

farm_map.save(tmp / "farm_map.html")
html = (tmp / "farm_map.html").read_text(encoding="utf-8")
print("HTML file saved:", "farm_map.html")
print("loads the Leaflet library:", "leaflet" in html.lower())
print("number of markers (Marker):", html.count("L.marker()"))
print("contains the plot name 'Meranti':", "Meranti" in html)

```

```

HTML file saved: farm_map.html
loads the Leaflet library: True
number of markers (Marker): 7
contains the plot name 'Meranti': True

```

- folium uses **EPSG:4326** (degrees), and its location order is [**latitude, longitude**]: the opposite of (x, y). Mixing up this order makes the map “land” in the sea.
- The script adds a plots layer (with a *tooltip* that appears when the cursor is near), the river, seven markers (four ponds and three pens), and a layer control.
- Evidence we can print: the file contains Leaflet, contains seven `L.marker()`, and contains the commodity name `Meranti`.

TECHNICAL STUFF (feel free to skip)

I did **not open** `farm_map.html` in a browser while making this book; only the file’s contents were checked, as in the output above. Note also that the base map (`OpenStreetMap`) is loaded from the internet when the map is opened, so without signal you see only the plot shapes with no background. Copying sensitive data into an HTML file means that data travels to anyone who receives the file.

PostGIS: a database that understands maps

PostgreSQL is a relational database (Chapter 24); **PostGIS** is its extension that adds the `geometry` type, hundreds of spatial functions (`ST_Area`, `ST_Distance`, ...), and **spatial indexes**. The advantages over files: many users at once, big data, and fast searching by location.

For this chapter, the book builder starts a **temporary PostgreSQL cluster** (random port, on this computer only, deleted afterwards) and gives its address through the `DATABASE_URL` variable. Scripts that use it are marked `@pg`. The author’s own PostgreSQL server is not touched. On your computer, the default

value `postgresql://postgres@localhost:5432/learn` is only an **example**: replace it with the address of your own server (and never put a password in code; read it from an environment variable, Chapter 34).

Creating tables and loading data

```
# Chapter 40 - PostGIS: creating spatial tables and loading data with psycopg
import os

import geopandas as gpd
import psycopg

# On your computer, replace this default address with the address of your own
PostgreSQL+PostGIS server
address = os.environ.get("DATABASE_URL", "postgresql://postgres@localhost:5432/
learn")

plots = gpd.read_file("data/plot.geojson")
ponds = gpd.read_file("data/pond.geojson")
rivers = gpd.read_file("data/river.geojson")

with psycopg.connect(address) as conn:
    conn.execute("CREATE EXTENSION IF NOT EXISTS postgis")
    version = conn.execute("SELECT PostGIS_Lib_Version()").fetchone()[0]
    print("PostGIS is active, major version:", version.split(".")[0])

    conn.execute("DROP TABLE IF EXISTS plot, pond, river")
    conn.execute("""CREATE TABLE plot (
        code text PRIMARY KEY, sector text, commodity text,
        geom geometry(Polygon, 4326)""") # geometry type AND SRID
are locked
    conn.execute("""CREATE TABLE pond (
        code text PRIMARY KEY, kind text, fish_count int,
        geom geometry(Polygon, 4326)""")

    conn.execute("""CREATE TABLE river (
        name text, geom geometry(LineString, 4326)""")

    for _, p in plots.iterrows(): # WKT = geometry as text
        conn.execute("INSERT INTO plot VALUES (%s, %s, %s, ST_GeomFromText(%s, 4326))",
            (p["code"], p["sector"], p["commodity"], p.geometry.wkt))
    for _, k in ponds.iterrows():
        conn.execute("INSERT INTO pond VALUES (%s, %s, %s, ST_GeomFromText(%s, 4326))",
            (k["code"], k["kind"], int(k["fish_count"]), k.geometry.wkt))

    conn.execute("INSERT INTO river VALUES (%s, ST_GeomFromText(%s, 4326))",
        (rivers["name"].iloc[0], rivers.geometry.iloc[0].wkt))

    # GiST spatial index: makes searches by location fast
    conn.execute("CREATE INDEX plot_geom_idx ON plot USING GIST (geom)")
    conn.execute("CREATE INDEX pond_geom_idx ON pond USING GIST (geom)")

    for table in ("plot", "pond", "river"):
        n = conn.execute(f"SELECT count(*) FROM {table}").fetchone()[0]
        print(f"rows in table {table}: ", n)
    print("geometry type of table plot:",
        conn.execute("SELECT DISTINCT ST_GeometryType(geom) FROM plot").fetchone()[0])
```

```
PostGIS is active, major version: 3
rows in table plot: 6
rows in table pond: 4
```

```
rows in table river: 1
geometry type of table plot: ST_Polygon
```

- `CREATE EXTENSION IF NOT EXISTS postgis` activates PostGIS in the database (it needs enough rights on your server).
- `geometry(Polygon, 4326)` locks the **shape type** and the **SRID** (the CRS number). Inserting another type is refused: a safeguard files do not have.
- Shapes are sent as **WKT** (*Well-Known Text*), text like `POLYGON((...))`, converted by `ST_GeomFromText`.
- A **GiST index** makes searching by location fast. Result: 6 plots, 4 ponds, 1 river.

Spatial queries

```
# Chapter 40 - Spatial queries in PostGIS (uses the tables from script 03)
import os

import psycopg

address = os.environ.get("DATABASE_URL", "postgresql://postgres@localhost:5432/learn")

with psycopg.connect(address) as conn:
    # Area: the geography type computes on the round earth, the result is in square metres
    print("plot area (ha):")
    for code, ha in conn.execute(
        "SELECT code, round((ST_Area(geom::geography) / 10000)::numeric, 2) FROM plot ORDER BY code"):
        print(f" {code}: {ha}")

    # Plots within 60 m of the river: ST_DWithin on geography (distance in metres)
    print("\nplots within 60 m of the river:")
    for (code,) in conn.execute(
        "SELECT p.code FROM plot p, river r "
        "WHERE ST_DWithin(p.geom::geography, r.geom::geography, 60) ORDER BY p.code"):
        print(" ", code)

    # Nearest ponds to the warehouse point (operator <-> = order by distance, uses the index)
    print("\nthree ponds nearest to the warehouse point:")
    warehouse = "ST_SetSRID(ST_MakePoint(106.8140, -6.9950), 4326)"
    for code, distance in conn.execute(
        f"SELECT code, round(ST_Distance(geom::geography, {warehouse}::geography)::numeric) "
        f"FROM pond ORDER BY geom <-> {warehouse} LIMIT 3"):
        print(f" pond {code}: {distance} m")

    # Joining two tables by location: plots and ponds that touch each other
    n = conn.execute("SELECT count(*) FROM plot p JOIN pond k ON ST_Intersects(p.geom, k.geom)").fetchone()[0]
    print("\nplot-pond pairs that overlap:", n)
```

```
plot area (ha):
H1: 9.00
H2: 6.92
K1: 4.00
K2: 3.00
S1: 3.00
S2: 2.25
```

```
plots within 60 m of the river:
```

```
K2
S1
S2
```

three ponds nearest to the warehouse point:

```
pond B: 82 m
pond C: 172 m
pond A: 274 m
```

plot-pond pairs that overlap: 0

Note:

- Using `::geography` makes PostGIS calculate on the **round earth** and answer in **metres** even though the data is in degrees. H2 shows 6.92 ha here while GeoPandas (UTM) calculated 6.93 ha: **two different methods give a small difference** because the UTM projection slightly changes sizes. For field purposes both are close enough; for legal purposes, follow the official rules.
- `ST_DWithin(..., 60)` finds plots within 60 m of the river: K2, S1, S2, the same as the Chapter 38 overlay.
- The `<->` operator sorts by distance and uses the index: the three ponds nearest to the warehouse point (an example point) are B (82 m), C (172 m), A (274 m).
- No plot overlaps a pond (result 0).

⚠ WARNING

Do not build SQL by pasting user text into a string (`f"... {input} ..."`): that opens the door to **SQL injection**. For **values**, always use `%s` parameters as in the `INSERT` of script 03. Script 04 pastes text only because the content is a fixed constant of our own, not from a user.

PostGIS and GeoPandas

```
# Chapter 40 - PostGIS and GeoPandas: read a query, write a new table
import os

import geopandas as gpd
from sqlalchemy import create_engine, text

address = os.environ.get("DATABASE_URL", "postgresql://postgres@localhost:5432/learn")
engine = create_engine(address.replace("postgresql://", "postgresql+psycopg://"))

# Read a query result straight into a GeoDataFrame
sql = "SELECT code, sector, geom FROM plot WHERE sector <> 'forest'"
not_forest = gpd.read_postgis(sql, engine, geom_col="geom")
print(not_forest[["code", "sector"]].to_string(index=False))
print("CRS read:", not_forest.crs)

# Write a GeoDataFrame to a new table (geoalchemy2 is needed behind the scenes)
pens = gpd.read_file("data/pen.geojson")
pens.to_postgis("pen", engine, if_exists="replace", index=False)
with engine.connect() as c:
    print("rows in table pen:", c.execute(text("SELECT count(*) FROM pen")).scalar())
    print("geometry column type:", c.execute(text(
        "SELECT type || ' / SRID ' || srid FROM geometry_columns WHERE f_table_name = 'pen'
    )).scalar())
```

```
code    sector
K1      orchard
K2      orchard
```

```

S1 ricefield
S2 ricefield
CRS read: EPSG:4326
rows in table pen: 3
geometry column type: POINT / SRID 4326

```

`read_postgis` turns a query result into a `GeoDataFrame`, with the CRS read automatically (EPSG:4326). `to_postgis` writes a `GeoDataFrame` to a new table and creates a geometry-typed column (POINT / SRID 4326). Both need **SQLAlchemy** and, for writing, **GeoAlchemy2** (I installed version 0.20.0 through `alat/pasang.py`).

PyQGIS: driving QGIS from Python

QGIS is a very popular free GIS program. It has its own Python and a module called **PyQGIS** (`qgis.core`, `qgis.analysis`) for automating work: loading layers, processing, styling, printing maps. How to use it: the script is run by **QGIS's own Python**, not this project's Python.

On the author's computer, QGIS 4.0.2 is installed in `C:\Program Files\QGIS 4.0.2` with its own Python 3.12. The command `bin\python-qgis.bat` prepares all the paths and then runs that Python. That is why the following scripts carry a `@python:` directive pointing at that file; **on your computer the path will differ** (or use the Python console inside QGIS).

The basic pattern of a script with no interface (following the official PyQGIS cookbook):

1. `QgsApplication.setPrefixPath("<qgis folder>", True)`: tell it where it is installed.
2. `app = QgsApplication([], False)`: `False = no GUI`.
3. `app.initQgis()`: load the data providers.
4. ... your work ...
5. `app.exitQgis()`: clean up.

Our addition: `os.environ["QT_QPA_PLATFORM"] = "offscreen"` before `QgsApplication` is created, so Qt does not look for a screen.

```

# Chapter 40 - PyQGIS without an interface: load a layer, read attributes
import os

os.environ["QT_QPA_PLATFORM"] = "offscreen"          # no window (must be set before
QgsApplication is created)

from qgis.core import Qgs, QgsApplication, QgsProject, QgsVectorLayer

QgsApplication.setPrefixPath("C:/Program Files/QGIS 4.0.2/apps/qgis", True)
app = QgsApplication([], False)                      # False = no GUI
app.initQgis()
print("QGIS major version:", Qgs.version().split(".")[0])

plots = QgsVectorLayer("data/plot.geojson", "plot", "ogr")
print("layer valid:", plots.isValid(), "| feature count:", plots.featureCount())
print("CRS:", plots.crs().authid(), "| geometry type:", plots.geometryType().name)
print("fields:", [f.name() for f in plots.fields()])

for feature in plots.getFeatures():
    if feature["sector"] == "ricefield":
        print("ricefield:", feature["code"], feature["commodity"])

QgsProject.instance().addMapLayer(plots)
print("layers in the project:", [l.name() for l in
QgsProject.instance().mapLayers().values()])
app.exitQgis()

```

```

QGIS major version: 4
layer valid: True | feature count: 6

```

```
CRS: EPSG:4326 | geometry type: Polygon
fields: ['code', 'sector', 'commodity']
ricefield: S1 rice
ricefield: S2 rice
layers in the project: ['plot']
```

The output proves the script ran on **QGIS 4** (major version 4): six features read, CRS EPSG:4326, polygon geometry type, and the fields `code`, `sector`, `commodity`. Ricefields S1 and S2 were found by filtering the attributes.

Processing algorithms, styling, and drawing a map

QGIS has hundreds of processing algorithms (*Processing*), e.g. `native:buffer` and `native:reprojectlayer`. From a script, call them with `processing.run(name, parameters)`.

```
# Chapter 40 - PyQGIS: Processing algorithms, styling, and drawing a map to PNG
import os

os.environ["QT_QPA_PLATFORM"] = "offscreen"

import sys
from pathlib import Path

from qgis.core import (QgsApplication, QgsCategorizedSymbolRenderer,
                        QgsCoordinateReferenceSystem,
                        QgsMapRendererSequentialJob, QgsMapSettings, QgsProject,
                        QgsRendererCategory,
                        QgsSymbol, QgsVectorLayer)
from qgis.PyQt.QtCore import QSize
from qgis.PyQt.QtGui import QColor

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
QgsApplication.setPrefixPath("C:/Program Files/QGIS 4.0.2/apps/qgis", True)
app = QgsApplication([], False)
app.initQgis()

# Processing: the `processing` module lives in the QGIS plugins folder, so we add its
path first
sys.path.append("C:/Program Files/QGIS 4.0.2/apps/qgis/python/plugins")
import processing
from processing.core.Processing import Processing
from qgis.analysis import QgsNativeAlgorithms

QgsApplication.processingRegistry().addProvider(QgsNativeAlgorithms())
Processing.initialize()

plots = QgsVectorLayer("data/plot.geojson", "plot", "ogr")
river = QgsVectorLayer("data/river.geojson", "river", "ogr")

flat = processing.run("native:reprojectlayer", {
    "INPUT": plots, "TARGET_CRS": QgsCoordinateReferenceSystem("EPSG:32748"), "OUTPUT":
    "memory:{}".format("OUTPUT")
})
print("plot areas in UTM (m2):", [round(f.geometry().area()) for f in
flat.getFeatures()])

river_utm = processing.run("native:reprojectlayer", {
    "INPUT": river, "TARGET_CRS": QgsCoordinateReferenceSystem("EPSG:32748"), "OUTPUT":
    "memory:{}".format("OUTPUT")
})
zone = processing.run("native:buffer", {"INPUT": river_utm, "DISTANCE": 60, "SEGMENTS":
8, "OUTPUT": "memory:{}".format("OUTPUT")})
```

```

print("features in the buffer zone:", zone.featureCount(), "| area (ha):",
      round(next(zone.getFeatures()).geometry().area() / 10_000, 2))

# Style: colour by the sector column
categories = []
for value, colour in [("forest", "#4f8a3b"), ("orchard", "#d9a441"), ("ricefield",
"#e8dc7a")]:
    symbol = QgsSymbol.defaultSymbol(flat.geometryType())
    symbol.setColor(QColor(colour))
    categories.append(QgsRendererCategory(value, symbol, value))
flat.setRenderer(QgsCategorizedSymbolRenderer("sector", categories))
zone.renderer().symbol().setColor(QColor(59, 106, 154, 90)) # transparent blue

# Draw the map into a PNG image
settings = QgsMapSettings()
settings.setLayers([zone, flat])
settings.setDestinationCrs(flat.crs())
settings.setExtent(zone.extent())
settings.setOutputSize(QSize(700, 450))
settings.setBackgroundColor(QColor("white"))
job = QgsMapRendererSequentialJob(settings)
job.start()
job.waitForFinished()
image = job.renderedImage()
print("image size:", image.width(), "x", image.height(), "| saved:",
      image.save(str(tmp / "qgis_map.png")))

project = QgsProject.instance()
project.setCrs(flat.crs())
project.addMapLayers([zone, flat])
print("QGIS project saved:", project.write(str(tmp / "farm.qgz")))
app.exitQgis()

```

```

plot areas in UTM (m2): [90001, 69252, 39999, 30000, 29999, 22500]
features in the buffer zone: 1 | area (ha): 13.23
image size: 700 x 450 | saved: True
QGIS project saved: True

```

The plot areas after conversion to UTM (90001, 69252, 39999, ... m²) match the GeoPandas calculation. The 60 m buffer zone covers 13.23 ha (GeoPandas 13.24 ha: slightly different because of the number of segments at the buffer ends). After that the script gives a **style** per category (colour by the sector column), draws a 700 × 450 pixel map into a PNG, and saves a **QGIS project** (.qgz) that can be opened in real QGIS.

i TIP

Because the `processing` module lives in QGIS's plugins folder, the script adds `.../apps/qgis/python/plugins` to `sys.path` and registers `QgsNativeAlgorithms` before `processing.run`. Without those two steps the algorithms are not found.

QGIS 4 differs from QGIS 3

QGIS 4.0 (released March 2026) moved from Qt5/PyQt5 to **Qt6/PyQt6**. As a result, much old PyQGIS code needs adjusting. The two changes that trip people most:

1. **Qt enums must be written in full:** `Qt.AlignLeft` becomes `Qt.AlignmentFlag.AlignLeft`.
2. **Field types use `QMetaType.Type`** (not `QVariant`) in `QgsField`.

```

# Chapter 40 - QGIS 4 (Qt6): Qt enums must be fully scoped, field types use QMetaType
import os

```

```

os.environ["QT_QPA_PLATFORM"] = "offscreen"

from qgis.core import Qgis, QgsField
from qgis.PyQt.QtCore import QMetaType, Qt

# QGIS 3 (PyQt5) style: Qt.AlignLeft. In QGIS 4 (PyQt6) that short name no longer exists.
try:
    Qt.AlignLeft
except AttributeError as error:
    print("old style fails ->", error)
print("new style Qt.AlignmentFlag.AlignLeft ->", int(Qt.AlignmentFlag.AlignLeft))

# Field types: QMetaType.Type replaces QVariant in QgsField
field = QgsField("fish_count", QMetaType.Type.Int)
print("field name:", field.name(), "| type:", field.type())

# QGIS's own enums are also best written in full
print("polygon geometry type:", Qgis.GeometryType.Polygon.name)

old style fails -> type object 'Qt' has no attribute 'AlignLeft'
new style Qt.AlignmentFlag.AlignLeft -> 1
field name: fish_count | type: Type.Int
polygon geometry type: Polygon

```

What I proved on QGIS 4.0.2 on this computer: the old style `Qt.AlignLeft` fails with `AttributeError`, the new style works, and `QgsField` with `QMetaType.Type.Int` works. Some old QGIS names still work through a compatibility layer (I tried `QgsWkbTypes.PolygonGeometry` and `QgsLayoutExporter.Success`, and both still answer), but the full form such as `Qgis.GeometryType.Polygon` is safer for the future. The full list of changes is in the Qt6 compatibility guide on the QGIS website; I did not read all of it, so consult that guide for your own code.

TECHNICAL STUFF (feel free to skip)

Two findings from experiments on the author's computer: (1) a print layout (`QgsPrintLayout`) can be exported to PNG without an interface, but **label text appears as black boxes** in `offscreen` mode (probably because no fonts were loaded; I did not check the exact cause). That is why this chapter's scripts use `QgsMapRendererSequentialJob` to draw the map, not a layout with text. (2) Exporting a layout prints a GDAL "ERROR 6" message in the output even though it succeeds; I did not use it so the output stays clean.

The Python and QGIS workflow

Need	Fitting tool
Repeated calculation and analysis	GeoPandas / Rasterio (your project's Python)
Shared storage, many users	PostGIS
Inspecting and editing maps by eye and mouse	QGIS (the interface)
Automating QGIS steps (print 20 similar maps)	PyQGIS
Maps for reports or books	Matplotlib
Maps to share on the web	folium

A common working pattern: data enters **PostGIS** or **GeoPackage**; analysts inspect it in **QGIS**; **GeoPandas** scripts calculate the summaries; and **PyQGIS** or **Matplotlib** prints the weekly map.

Running Project I5: location data in two places

We save the five location layers (plots, ponds, pens, river, hamlets) to **PostGIS** (to be shared and later read by the API) **and** to **GeoPackage** (one file that can be sent by email and opened in QGIS), then check that both hold the same thing.

```
# Chapter 40 - Running project I5: location data stored in PostGIS and GeoPackage
import os
from pathlib import Path

import geopandas as gpd
import pandas as pd
from sqlalchemy import create_engine

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
address = os.environ.get("DATABASE_URL", "postgresql://postgres@localhost:5432/learn")
engine = create_engine(address.replace("postgresql://", "postgresql+psycopg://"))

layers = ["plot", "pond", "pen", "river", "hamlet"]
gpkg = tmp / "farm.gpkg"
gpkg.unlink(missing_ok=True)
for name in layers:
    gdf = gpd.read_file(f"data/{name}.geojson")
    gdf.to_postgis(f"farm_{name}", engine, if_exists="replace", index=False) #
server
    gdf.to_file(gpkg, layer=name, driver="GPKG") #
portable file

print("layer    PostGIS    GeoPackage")
for name in layers:
    a = len(gpd.read_postgis(f"SELECT * FROM farm_{name}", engine, geom_col="geometry"))
    b = len(gpd.read_file(gpkg, layer=name))
    print(f"{{name:8s}} {{a:7d}} {{b:11d}}")

# Area per sector: computed on the PostGIS server and in Python from the GeoPackage;
# they must agree
from_server = pd.read_sql(
    "SELECT sector, round((sum(ST_Area(geometry)::geography)) / 10000)::numeric, 1) AS
ha "
    "FROM farm_plot GROUP BY sector ORDER BY sector", engine)
print("\narea per sector (ha) from PostGIS:")
print(from_server.to_string(index=False))
plots = gpd.read_file(gpkg, layer="plot").to_crs(32748)
print("\narea per sector (ha) from GeoPackage:")
print(plots.assign(ha=plots.area
10_000).groupby("sector")["ha"].sum().round(1).to_string()) /
```

layer	PostGIS	GeoPackage
plot	6	6
pond	4	4
pen	3	3
river	1	1
hamlet	2	2

```
area per sector (ha) from PostGIS:
  sector  ha
  forest 15.9
  orchard  7.0
  ricefield 5.2
```



```
area per sector (ha) from GeoPackage:
sector
forest      15.9
orchard      7.0
ricefield    5.2
```

Five layers, the same row counts in both places, and matching area per sector: forest 15.9 ha, orchard 7.0 ha, ricefield 5.2 ha (all rounded to one decimal; ricefield 5.25 becomes 5.2 through Python's rounding). The PostGIS tables get the prefix `farm_` so they do not clash with the tables of script 03. That ends the map part of the Series I5 project: the location data is tidy and stored.

Exercises

1. In `01_static_map.py`, add a simple scale bar (a 100 m line with the text "100 m") at the bottom right of the map.
2. In `02_folium_map.py`, change `tiles="OpenStreetMap"` to `tiles="CartoDB positron"` and see how the file content differs.
3. In `04_postgis_spatial_queries.py`, add a query that counts the fish per hamlet using `ST_Within` between the pond centre point (`ST_Centroid`) and the hamlet polygon. (The hamlet table is not loaded yet: add it in script 03.)
4. **Challenge:** in `07_pyqgis_process_and_render.py`, add the algorithm `native:intersection` between plots and the zone, then print the cut area of each plot, and compare with Chapter 38.

Chapter Summary

- **Matplotlib** for static maps (don't forget `set_aspect("equal")`); **folium** for interactive maps (HTML, order [latitude, longitude]).
- **PostGIS** adds the `geometry` type, `ST_*` functions, and spatial indexes to PostgreSQL; send values through parameters, not pasted text.
- `::geography` calculates on the round earth in metres; the result can differ slightly from a UTM calculation.
- **PyQGIS** runs in QGIS's own Python; a script with no interface needs `setPrefixPath`, `QgsApplication([], False)`, `initQgis`, and `exitQgis`.
- **QGIS 4** uses Qt6: write Qt enums in full and use `QMetaType.Type`.
- Double storage: PostGIS for sharing, GeoPackage for portability.

Next, in Chapter 41: task automation and web scraping.

Chapter 41 — Automation and Scripting: Let Python Work by Itself

Series I5 — First Applications: Web, Maps, and Dashboards

In this chapter, you will:

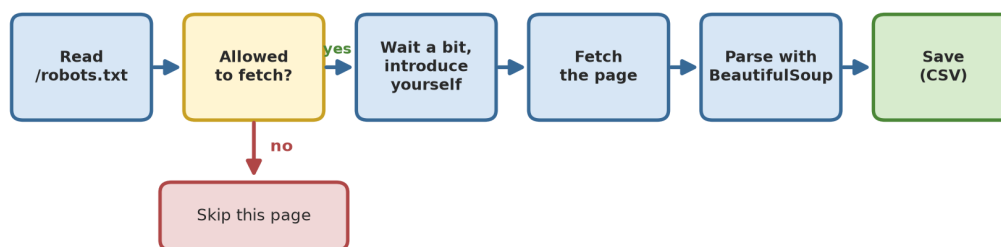
- Run other programs from Python with **subprocess**
- Automate file chores and service checks (infrastructure)
- Understand how to schedule scripts on Windows (Task Scheduler) and Linux/macOS (cron)
- Read web pages with **BeautifulSoup** and **Scrapy**, with the right ethics
- Build the weekly report automation: the Series I5 running project

All scripts for this chapter are in the `code/ch41/` folder. On the web version, you can also [download them all at once](#) (zip).

In Chapters 36 to 40 you built web applications and maps. Now imagine a repeating job: every Monday morning someone collects the numbers from each business unit, tidies them, and writes a report. That is like an officer who goes around every week with a notebook. Python can be an **officer who never forgets and never gets bored**. A simple rule of thumb: **when you do the same thing for the third time, write a script**.

This chapter has two big parts. First, **automation**: running programs, tidying files, checking services, and scheduling it all. Second, **web scraping**: reading data from web pages. For the second there is a hard rule in this book: we only practise on **self-made sites running on our own computer**, and we discuss the ethics first.

Polite scraping: the order of the steps



Only your own site or one that clearly allows it. robots.txt is a request, not a lock.

Figure 0.1: Polite scraping: the order of the steps.

subprocess: calling another program

Python has the **subprocess** module to run another program (the **child process**) and wait for its result. It is like asking an assistant to do something and report back.

```
# Chapter 41 - subprocess: running another program from Python
import subprocess
import sys

# sys.executable = path of the Python we are using; "-c" = run a short piece of code
```

```

result = subprocess.run(
    [sys.executable, "-c", "print('hello from the child process')"],
    capture_output=True, text=True, encoding="utf-8",
)
print("exit code   :", result.returncode)
print("stdout      :", result.stdout.strip())
print("stderr empty:", result.stderr == "")

# Arguments are passed as a LIST, one item per word (no shell)
command = [sys.executable, "-c", "import sys; print(' + '.join(sys.argv[1:]))", "rice",
"corn"]
result = subprocess.run(command, capture_output=True, text=True, encoding="utf-8")
print("arguments received:", result.stdout.strip())

```

```

exit code   : 0
stdout      : hello from the child process
stderr empty: True
arguments received: rice + corn

```

- `sys.executable` is the path of the Python being used, so the child uses the same Python. `-c` means “run this short piece of code”.
- `capture_output=True` catches what the child prints (`stdout`) and its complaints (`stderr`). `text=True` makes the result text instead of bytes.
- The **exit code** (`returncode`) 0 means success. Any other number means failure.
- Arguments are sent as a **list**, one item per word. According to Python’s official documentation, this way the module itself takes care of quotes and spaces in file names.

⚠ WARNING

Do not be tempted to use `shell=True` with text that comes from a user or a file. Python’s documentation warns that with `shell=True` it is you who must secure every space and special character; otherwise people can slip in other commands (**shell injection**). With a list, special characters are passed to the child as they are, and that is safe.

A child program can fail in three ways. All three can be handled:

```

# Chapter 41 - subprocess: three ways to fail and how to handle them
import subprocess
import sys

# 1. The child program ends with an error; check=True turns that into an exception
try:
    subprocess.run([sys.executable, "-c", "raise SystemExit(3)"], check=True)
except subprocess.CalledProcessError as e:
    print("1. exit code is not 0:", e.returncode)

# 2. The child program takes too long; timeout stops the waiting
try:
    subprocess.run([sys.executable, "-c", "import time; time.sleep(5)"], timeout=0.5)
except subprocess.TimeoutExpired:
    print("2. passed the 0.5 second limit, child process stopped")

# 3. The program does not exist at all
try:
    subprocess.run(["program-that-does-not-exist-123"])
except FileNotFoundError:
    print("3. program not found")

```

1. exit code is not 0: 3
2. passed the 0.5 second limit, child process stopped
3. program not found

`check=True` turns a non-zero exit code into a `CalledProcessError` exception. `timeout` stops a child that takes too long and raises `TimeoutExpired`. A missing program raises `FileNotFoundError`. A good automation script **always thinks about all three**, because nobody is watching the screen when the script runs at three in the morning.

File automation: tidying a folder

A classic chore: the downloads folder fills up with all kinds of files. This script moves each file to a folder according to its extension.

```
# Chapter 41 - Automating a daily chore: tidying a downloads folder by file type
import shutil
from pathlib import Path

work = Path(__file__).parent / "_tmp" / "downloads"
shutil.rmtree(work, ignore_errors=True)
work.mkdir(parents=True)
for name in ["harvest_a.csv", "harvest_b.csv", "pond_map.png", "report_may.pdf",
             "notes.txt", "orchard_photo.png"]:
    (work / name).write_text("sample content", encoding="utf-8")

FOLDER_MAP = {".csv": "data", ".png": "images", ".pdf": "documents"}

moved = 0
for file in sorted(work.iterdir()):
    if not file.is_file():
        continue
    target = work / FOLDER_MAP.get(file.suffix, "other")
    target.mkdir(exist_ok=True)
    shutil.move(str(file), str(target / file.name))
    moved += 1

print("files moved:", moved)
for folder in sorted(p for p in work.iterdir() if p.is_dir()):
    print(f"{folder.name:10}", sorted(p.name for p in folder.iterdir()))
```

```
files moved: 6
data        ['harvest_a.csv', 'harvest_b.csv']
documents   ['report_may.pdf']
images      ['orchard_photo.png', 'pond_map.png']
other       ['notes.txt']
```

Notice two good habits. First, the script works in its own `_tmp` folder, not in your real folders, so it is safe to try. Second, `FOLDER_MAP.get(file.suffix, "other")` provides a fallback folder for unknown types (`notes.txt` goes to `other`), so no file is left behind.

TIP

Before moving or deleting files automatically, first run a version that **only prints** what it would do. Once the list looks right, swap in the real commands.

Infrastructure automation: checking services

Infrastructure is all the “machines behind the scenes” that run your application: servers, web services, databases, disk space. Infrastructure automation means writing scripts that check and look after them. The most basic example is a **health check**: ask for the home page, then see whether the answer is healthy.

```
# Chapter 41 - Infrastructure automation: a service health checker
import shutil
import urllib.error
import urllib.request

from local_site import LocalSite

def check(address):
    """Returns a status text: 'healthy', 'HTTP error <code>', or 'not connected'."""
    try:
        with urllib.request.urlopen(address, timeout=2) as response:
            return "healthy" if response.status == 200 else f"status {response.status}"
    except urllib.error.HTTPError as e:
        return f"HTTP error {e.code}"
    except urllib.error.URLError:
        return "not connected"

with LocalSite() as site:
    address = site.url
    print("home page      :", check(address + "/"))
    print("missing page   :", check(address + "/does-not-exist"))

# After the `with` block the server is switched off: the checker must notice
print("after shutdown :", check(address + "/"))

# Another common check: free disk space (only True/False is printed to keep the
output stable)
print("disk has room  :", shutil.disk_usage(".").free > 100 * 1024 * 1024)

home page      : healthy
missing page   : HTTP error 404
after shutdown : not connected
disk has room  : True
```

The server being checked here is a small server we made ourselves on this computer (the helper module `local_site.py`, explained in the scraping part). Three different results deserve to be told apart: **healthy** (code 200), **HTTP error** (the server is alive but the page does not exist, 404), and **not connected** (the server is down). In practice, a script like this is scheduled every few minutes and sends an alert when the result is not “healthy”. Sending alerts (email or messages) is not practised here because it needs services outside this computer.

Scheduling scripts

An automation script is only useful when it **runs by itself at the right time**. Python does not schedule itself: that job is handed to the operating system’s scheduler.

System	Scheduler	How
Windows	Task Scheduler	the <code>schtasks /create</code> command, or the graphical Task Scheduler app

System	Scheduler	How
Linux and macOS	cron	a line in the crontab file (crontab -e)

```
# Chapter 41 - Scheduling a script: the commands are ONLY printed, never run
from pathlib import Path

# Windows Task Scheduler: one task = one program, so we wrap the call in a .bat file
bat = "\n".join([
    "@echo off",
    r"C:\project\.venv\Scripts\python.exe C:\project\weekly_report.py",
    "",
])
target = Path(__file__).parent / "_tmp" / "run_report.bat"
target.parent.mkdir(exist_ok=True)
target.write_text(bat, encoding="ascii")
print("contents of run_report.bat:")
print(bat)

print("Windows command (run it yourself in your terminal):")
print(r"schtasks /create /tn WeeklyReport /tr C:\project\run_report.bat /sc weekly /d MON /st 07:00")

# cron (Linux/macOS): five time columns, then the command
cron = "0 7 * * 1 /home/you/project/.venv/bin/python /home/you/project/weekly_report.py"
print("\ncrontab line (paste it through `crontab -e`):")
print(cron)

LIMITS = [("minute", 0, 59), ("hour", 0, 23), ("day of month", 1, 31), ("month", 1, 12), ("weekday", 0, 6)]

def check_cron(line):
    """A simple checker: only '*' or one number per column, and the number must be in range."""
    columns = line.split()[:5]
    for text, (name, low, high) in zip(columns, LIMITS):
        if text != "*" and not (text.isdigit() and low <= int(text) <= high):
            return f"column {name} is wrong: {text}"
    return "the five time columns look sensible"

print("\nCheck:", check_cron(cron))
print("Wrong example:", check_cron("0 25 * * 1 command"))
```

```
contents of run_report.bat:
@echo off
C:\project\.venv\Scripts\python.exe C:\project\weekly_report.py

Windows command (run it yourself in your terminal):
schtasks /create /tn WeeklyReport /tr C:\project\run_report.bat /sc weekly /d MON /st 07:00

crontab line (paste it through `crontab -e`):
0 7 * * 1 /home/you/project/.venv/bin/python /home/you/project/weekly_report.py

Check: the five time columns look sensible
Wrong example: column hour is wrong: 25
```

This code only prints the commands. The scheduling commands are not run in this book because they change your computer's settings. You run them yourself when you are ready. The explanation, according to the official documentation:

- **Windows (schtasks):** `/tn` names the task, `/tr` sets the program to run, `/sc weekly` means weekly, `/d MON` Monday, `/st 07:00` the start time (24-hour format). One task runs only **one program**, so the documentation suggests a batch file (`.bat`) to run several steps. That is why we wrap Python in `run_report.bat`.
- **cron:** five time columns in order: minute (0-59), hour (0-23), day of month (1-31), month (1-12), day of week (0-6, with 0 = Sunday). So `0 7 * * 1` means **07:00 every Monday**. The `*` means "every value".

i REMEMBER

Three scheduling habits that will save you: (1) use **full paths** to Python and to the script, because the scheduler does not run from your folder; (2) always **write a log** to a file, because there is no screen to read; (3) **try it by hand first** before scheduling it.

Web scraping: ethics first, code second

Web scraping is reading web pages with a program, then picking out the useful parts (for example the contents of a table). The technique is easy. What is hard, and far more important, is **behaving properly**. Before writing a single line, hold on to these rules:

Question	What you must do
Is there an official way?	Look for an API or an official data download first. That is always better than scraping.
What does <code>robots.txt</code> say?	Read <code>/robots.txt</code> on that site and obey the parts that forbid.
What do the terms of use say?	Read the site's terms of service. If they forbid scraping, do not do it.
How often do I ask?	Leave a pause between requests and do not flood the server (rate limit).
Who am I to the server?	Introduce yourself with an honest User-Agent that can be contacted.
What is the data?	Be careful with personal data and copyright. Data you may read is not always data you may share.

`robots.txt` is described in the **RFC 9309** standard (the Robots Exclusion Protocol). The standard stresses that its rules are **not a form of access authorization**: it is a polite request obeyed by well-behaved crawlers, not a lock. That means nothing stops your program from breaking it. What stops it is **your attitude**. The standard also does not define `Crawl-delay`; that is an extra habit understood by some crawlers.

The rule of this book: every scraping exercise here runs against **self-made** HTML pages served by a server on your computer (`127.0.0.1`, an address only this computer can reach). We do not fetch pages from any real site. That is on purpose: we can practise fully without burdening anyone.

The practice site is made by the helper module `local_site.py` (in the script folder): one home page listing the sectors, five sector pages that each hold a table of business units, one "internal" page that `robots.txt` forbids, and the `robots.txt` file itself. The data is our sample business-unit data, **all numbers fictional**.

BeautifulSoup: reading HTML

BeautifulSoup (the `beautifulsoup4` package) turns HTML text into a tree you can walk through. Python's built-in parser is called `html.parser`; there are also `lxml` and `html5lib`, installed separately.

```
# Chapter 41 - BeautifulSoup: reading an HTML page (a self-made site on your own
computer)
import urllib.request

from bs4 import BeautifulSoup

from local_site import LocalSite

with LocalSite() as site:
    with urllib.request.urlopen(site.url + "/sector/pond.html", timeout=5) as r:
        html = r.read().decode("utf-8")

print("--- first 8 lines of the HTML ---")
print("\n".join(html.splitlines()[:8]))
print("...")

soup = BeautifulSoup(html, "html.parser")
print("\npage title :", soup.title.get_text())
print("main heading:", soup.h1.get_text())

# CSS selector: find every table row that has a data-id attribute
total = 0.0
for tr in soup.select("table#unit tbody tr[data-id]"):
    cells = [td.get_text(strip=True) for td in tr.find_all("td")]
    print(tr["data-id"], cells, "lat =", tr["data-lat"])
    total += float(tr.select_one("td.value").get_text())
print("total value :", total)

# Trap: find() returns None when nothing matches
table = soup.find("table", id="no-such-table")
print("\ntable we looked for:", table)
try:
    table.find_all("tr")
except AttributeError as e:
    print("error:", e)
```

```
--- first 8 lines of the HTML ---
<!doctype html>
<html lang="en">
<head><meta charset="utf-8"><title>Sector pond</title></head>
<body>
  <h1>Sector pond</h1>
  <table id="unit">
    <thead><tr><th>Unit</th><th>Kind</th><th>Value</th><th>Unit of measure</th></tr></thead>
    <tbody>
  ...

page title : Sector pond
main heading: Sector pond
L1 ['Pond A', 'tilapia', '432.4', 'kg'] lat = -6.644
L2 ['Pond B', 'catfish', '380.0', 'kg'] lat = -6.6452
L3 ['Pond C', 'tilapia', '295.5', 'kg'] lat = -6.6466
total value : 1107.9

table we looked for: None
error: 'NoneType' object has no attribute 'find_all'
```

Walk through the result:

- The first eight lines of HTML show the shape of the page: a `<table id="unit">` with a head and a body.

- `select("table#unit tbody tr[data-id]")` uses a **CSS selector**: “rows inside the body of the `unit` table that have a `data-id` attribute”. `select()` returns a list of all matches; `select_one()` only the first.
- `get_text(strip=True)` takes the text without extra spaces. Attributes such as `data-lat` are read like a dictionary: `tr["data-lat"]`.
- The total of the three ponds is $432.4 + 380.0 + 295.5 = 1107.9$ kg, matching the output.
- **Trap**: when the element you look for is missing, `find()` returns `None` (while `find_all()` returns an empty list). Calling a method on `None` raises `AttributeError`. That is why good code checks for `None` and **complains clearly** when the page changes shape.

robots.txt in code

Python has a built-in `robots.txt` reader: `urllib.robotparser`.

```
# Chapter 41 - robots.txt: reading a site's rules before fetching pages
import urllib.request
import urllib.robotparser

from local_site import LocalSite

AGENT = "BookPythonBot/1.0"

with LocalSite() as site:
    with urllib.request.urlopen(site.url + "/robots.txt", timeout=5) as r:
        content = r.read().decode("utf-8")
        print("contents of robots.txt:")
        print(content)

    robot = urllib.robotparser.RobotFileParser()
    robot.parse(content.splitlines())
    for path in ["/", "/sector/pond.html", "/private/notes.html"]:
        allowed = robot.can_fetch(AGENT, site.url + path)
        print(f"{path:24} allowed to fetch? {allowed}")
    print("Crawl-delay:", robot.crawl_delay(AGENT), "second(s)")

# robots.txt is only a request, not a lock: the server still answers if we force it
with urllib.request.urlopen(site.url + "/private/notes.html", timeout=5) as r:
    print("\nif forced, the server answers:", r.status, "(we must STILL NOT do it)")
```

```
contents of robots.txt:
User-agent: *
Disallow: /private/
Crawl-delay: 1

/                  allowed to fetch? True
/sector/pond.html  allowed to fetch? True
/private/notes.html allowed to fetch? False
Crawl-delay: 1 second(s)

if forced, the server answers: 200 (we must STILL NOT do it)
```

`can_fetch(agent, address)` answers whether our agent may fetch that address. The `/private/` page is forbidden (`False`). `crawl_delay()` returns the pause the site asks for (1 second), or `None` when there is none. Notice the last line: **the server still answers 200 if we force it**. That is exactly the RFC 9309 message from above: `robots.txt` is not a lock, so obedience is entirely up to us.

A polite scraper

Now we combine everything into one class: read `robots.txt` first, pause, and introduce itself.

```

# Chapter 41 - Helper module: a polite page fetcher (obeys robots.txt, waits, introduces
itself)
import time
import urllib.error
import urllib.request
import urllib.robotparser

from bs4 import BeautifulSoup

AGENT = "BookPythonBot/1.0 (learning example; own local site)"

def read_table(html):
    """Reads the unit table from one sector page into a list of dicts."""
    soup = BeautifulSoup(html, "html.parser")
    table = soup.find("table", id="unit")
    if table is None:
        # the page changed shape: do not silently
get it wrong
        raise ValueError("table #unit not found")
    result = []
    for tr in table.select("tbody tr"):
        cells = [td.get_text(strip=True) for td in tr.find_all("td")]
        result.append({
            "id": tr["data-id"], "unit": cells[0], "kind": cells[1],
            "lat": float(tr["data-lat"]), "lon": float(tr["data-lon"]),
            "value": float(cells[2]), "uom": cells[3],
        })
    return result

class PoliteFetcher:
    def __init__(self, base, agent=AGENT, default_delay=1.0, timeout=5):
        self.base, self.agent, self.timeout = base, agent, timeout
        self.refused = [] # addresses not fetched because of robots.txt
        self._last = 0.0
        self.robot = urllib.robotparser.RobotFileParser()
        text = self._download("/robots.txt")
        self.robot.parse(text.splitlines() if text else [])
        # Delay: follow Crawl-delay if present (a custom, not part of the standard),
otherwise the default
        self.delay = self.robot.crawl_delay(agent) or default_delay

    def _download(self, path):
        request = urllib.request.Request(self.base + path, headers={"User-Agent":
self.agent})
        try:
            with urllib.request.urlopen(request, timeout=self.timeout) as r:
                return r.read().decode("utf-8")
        except urllib.error.HTTPError:
            return ""

    def fetch(self, path):
        """Returns the HTML, or None if robots.txt forbids it."""
        if not self.robot.can_fetch(self.agent, self.base + path):
            self.refused.append(path)
            return None
        wait = self.delay - (time.monotonic() - self._last)
        if wait > 0:
            time.sleep(wait)
        try:

```

```

        return self._download(path)
    finally:
        self._last = time.monotonic()

```

Important points in this class:

- An honest **User-Agent**: BookPythonBot/1.0 (learning example; own local site). On a real site, include a way to contact you.
- `self.delay`: uses the site's Crawl-delay if there is one, otherwise 1 second.
- `fetch()` **refuses** addresses that `robots.txt` forbids (recorded in `refused`) and waits until the pause is over before asking again.
- `read_table()` raises `ValueError` when the table is not found.

The script that uses it:

```

# Chapter 41 - A polite scraper: obeys robots.txt, waits between requests, introduces
itself
import csv
from pathlib import Path

from bs4 import BeautifulSoup

from fetcher import PoliteFetcher, read_table
from local_site import LocalSite
from unit_data import read_units

with LocalSite() as site:
    bot = PoliteFetcher(site.url)
    print("delay used:", bot.delay, "second")

    index = BeautifulSoup(bot.fetch("/"), "html.parser")
    links = [a["href"] for a in index.select("#menu a")]
    print("links on the home page:", len(links))

    everything = []
    for path in links:
        html = bot.fetch(path)
        if html is None:
            print("skipped (robots.txt forbids it):", path)
            continue
        everything += read_table(html)
    log = list(site.log)

out = Path(__file__).parent / "_tmp" / "units_scraped.csv"
out.parent.mkdir(exist_ok=True)
with open(out, "w", encoding="utf-8", newline="") as f:
    writer = csv.DictWriter(f, fieldnames=list(everything[0]))
    writer.writeheader()
    writer.writerows(everything)

print("scraped rows:", len(everything))
print("same as the source data:", everything == [{k: u[k] for k in everything[0]} for
u in read_units()])
print("requests that reached the server:", len(log))
print("forbidden pages requested      :", [p for p, _ in log if p.startswith("/
private")])
print("self-introduction (User-Agent) :", sorted({ua for _, ua in log}))

```

```

delay used: 1 second
links on the home page: 6
skipped (robots.txt forbids it): /private/notes.html

```

```

scraped rows: 15
same as the source data: True
requests that reached the server: 7
forbidden pages requested      : []
self-introduction (User-Agent) : ['BookPythonBot/1.0 (learning example; own local
site)']

```

Let us check the numbers. The home page has **6 links**: five sectors and one internal page. The internal one is **skipped** because it is forbidden. The result is **15 rows**, exactly equal to the source data (**True**). The server received **7 requests**: 1 for `robots.txt`, 1 for the home page, and 5 for the sector pages. The list of forbidden pages requested is empty, and our self-introduction is honestly recorded on the server.

TECHNICAL STUFF (feel free to skip)

This script waits 1 second before each request, so it runs for a few seconds. That is on purpose and not slowness: it is the rate limit. On real sites, a reasonable pause is often longer, and large-scale scraping should be allowed in writing by the site owner.

Scrapy: a framework for big scraping jobs

Scrapy is a complete crawling **framework**: it manages the queue of addresses, pauses, concurrency, `robots.txt` obedience, and saving results. If BeautifulSoup is a **knife**, Scrapy is a **small factory**. For one or two pages BeautifulSoup is enough; for hundreds of pages on a routine, Scrapy is tidier.

	BeautifulSoup	Scrapy
Nature	HTML reading library	full crawling framework
Fetches pages?	no (use <code>urllib</code> or <code>httpx</code>)	yes
Pause and <code>robots.txt</code>	you write it	settings (DOWNLOAD_DELAY, ROBOTSTXT_OBEY)
Good for	small jobs	large, repeating crawl projects

Scrapy 2.19.0 is installed on the author's computer through `alat/pasang.py` for Python 3.14, and the script below ran successfully. (On your computer the installation is `pip install scrapy`.) Here is a small **spider** (crawler):

```

# Chapter 41 - Scrapy: a scraping framework, run against a self-made local site
import json
from pathlib import Path

import scrapy
from scrapy.crawler import CrawlerProcess

from local_site import LocalSite

OUT = Path(__file__).parent / "_tmp" / "scrapy_units.json"
OUT.parent.mkdir(exist_ok=True)

class UnitLab(scrapy.Spider):
    name = "unit_lab"

    def __init__(self, base, **kw):
        super().__init__(**kw)
        self.base = base

    async def start(self):
        yield scrapy.Request(self.base + "/")

```

```

def parse(self, response):
    # the menu has 5 sector links + 1 internal link that robots.txt forbids
    for a in response.css("#menu a"):
        yield response.follow(a, callback=self.parse_sector)

def parse_sector(self, response):
    for tr in response.css("table#unit tbody tr"):
        yield {
            "id": tr.attrib["data-id"],
            "unit": tr.css("td:nth-child(1)::text").get(),
            "value": float(tr.css("td.value::text").get()),
        }

with LocalSite() as site:
    process = CrawlerProcess(settings={
        "USER_AGENT": "BookPythonBot/1.0 (learning example; own local site)",
        "ROBOTSTXT_OBEY": True,
        "DOWNLOAD_DELAY": 1,
        "CONCURRENT_REQUESTS_PER_DOMAIN": 1,
        "LOG_LEVEL": "ERROR",
        "FEEDS": {str(OUT): {"format": "json", "overwrite": True}},
    })
    crawler = process.create_crawler(UnitLab)
    process.crawl(crawler, base=site.url)
    process.start()
    stats = crawler.stats.get_stats()

items = json.loads(OUT.read_text(encoding="utf-8"))
print("items collected          :", len(items))
print("first item (by id)       :", sorted(items, key=lambda x: x["id"])[0])
print("forbidden by robots.txt :", stats.get("robotstxt/forbidden"))
print("pages fetched (200)      :", stats.get("downloader/response_status_count/200"))

```

```

items collected          : 15
first item (by id)       : {'id': 'D1', 'unit': 'Pen 1', 'value': 350.0}
forbidden by robots.txt : 1
pages fetched (200)      : 7

```

How to read it:

- `start()` gives the starting address. `parse()` receives the home page and follows each menu link with `response.follow`. `parse_sector()` yields one **item** (a dictionary) for each table row.
- The settings in `CrawlerProcess` carry our ethics: `ROBOTSTXT_OBEY` turns on `robots.txt` obedience (the Scrapy documentation notes that projects made by scrapy `startproject` turn it on, but we write it explicitly), `DOWNLOAD_DELAY` gives a 1-second pause, `CONCURRENT_REQUESTS_PER_DOMAIN` limits simultaneous requests to one site, and `USER_AGENT` introduces us. The Scrapy documentation recommends a User-Agent that names your project and how to reach you.
- The result: **15 items**, **1** request refused by `robots.txt` (the internal page), and **7** pages with status 200: `robots.txt`, the home page, and five sector pages. This matches the count of our own scraper.

i REMEMBER

Scrapy has a **random jitter** on its pause: by default the delay is varied by about half to one and a half times `DOWNLOAD_DELAY`, so it is not too regular. That is why run time is not printed in this book, only stable counts.

Running Project I5: weekly report automation

Now all the ingredients come together. This project script fetches the data of all sectors from the local site with the polite fetcher, summarizes per sector, compares with last week's means, writes a Markdown report, and keeps a log.

```
# Chapter 41 - Running project I5: automating the weekly report (FICTIONAL numbers)
import logging
from pathlib import Path

from fetcher import PoliteFetcher, read_table
from local_site import LocalSite
from unit_data import summarize_by_sector

OUT = Path(__file__).parent / "_tmp"
OUT.mkdir(exist_ok=True)
logging.basicConfig(filename=OUT / "report.log", filemode="w", level=logging.INFO,
                    format="%(levelname)s %(message)s")

# Last week's means (sample numbers) to compare against
LAST_WEEK = {"forest": 34.8, "plantation": 250.0, "paddy": 1100.0, "pond": 360.0,
             "barn": 380.0}

def collect(base):
    bot = PoliteFetcher(base)
    units = []
    for sector in LAST_WEEK:
        html = bot.fetch(f"/sector/{sector}.html")
        if html is None:
            logging.warning("skipped: %s", sector)
            continue
        rows = [dict(u, sector=sector) for u in read_table(html)]
        units += rows
        logging.info("sector %s: %d units", sector, len(rows))
    return units

def build_report(units):
    lines = ["# Weekly report (fictional data)", "",
            "| Sector | Units | Mean | Unit | Change from last week |",
            "|---|---|---|---|---|"]
    for sector, r in summarize_by_sector(units).items():
        change = (r["mean"] - LAST_WEEK[sector]) / LAST_WEEK[sector] * 100
        lines.append(f"| {sector} | {r['count']} | {r['mean']:.1f} | {r['uom']} | {change:+.1f}% |")
    return "\n".join(lines) + "\n"

def main():
    with LocalSite() as site:
        units = collect(site.url)
        text = build_report(units)
        (OUT / "weekly_report.md").write_text(text, encoding="utf-8")
        logging.info("report finished: %d units", len(units))
    return text

if __name__ == "__main__":
    print(main())
    print("log lines:", len((OUT / "report.log").read_text().splitlines()))
```

```
# Weekly report (fictional data)

| Sector | Units | Mean | Unit | Change from last week |
|---|---|---|---|---|
| forest | 3 | 35.2 | cm | +1.2% |
| plantation | 4 | 257.5 | kg | +3.0% |
| paddy | 2 | 1115.2 | kg | +1.4% |
| pond | 3 | 369.3 | kg | +2.6% |
| barn | 3 | 383.8 | kg | +1.0% |

log lines: 6
```

Follow one row: the **pond** sector has 3 units with mean $(432.4 + 380.0 + 295.5) / 3 = 369.3$ kg. Last week's mean (a sample number) was 360.0, so $(369.3 - 360.0) / 360.0 = +2.6\%$. The script also writes `report.log` with 6 lines (5 sectors and 1 “report finished” line), proof that it can be monitored without a screen.

The script's structure is made to be schedulable: all the work is inside the `main()` function, and the `if __name__ == "__main__":` part only calls it. With the `.bat` wrapper or the cron line from the scheduling part, this report will assemble itself every Monday at 07:00. The sector means for forest (cm) and the others (kg) must not be added across sectors; that is why the report is made per sector.

Exercises

1. Change `01_subprocess_basics.py` so that the child process prints the sum of two numbers that you send as arguments, then read the result in the parent.
2. Add the file type `.xlsx` (folder sheets) to `03_file_automation.py`. How many files land in `other` now?
3. In `07_robots_ethics.py`, change `Disallow` in the `local_site.py` module (your own copy) so that `/sector/` is forbidden too. Make sure `can_fetch` becomes `False` for the sector pages.
4. **Challenge:** add a “top unit” column to the weekly report (the name of the unit with the largest value in each sector), then write the `schtasks` or cron line to schedule it for Friday at 16:00 (just print it, do not run it).

Chapter Summary

- `subprocess.run(..., capture_output=True, text=True)` runs another program; send arguments as a `list`, avoid `shell=True` with untrusted text.
- Handle three failures: exit code (`check=True`), timeout (`timeout`), missing program (`FileNotFoundError`).
- Scheduling is handed to **Task Scheduler** (Windows) or **cron** (Linux/macOS); use full paths, write a log, test by hand first.
- **Scraping:** look for an official API first; obey `robots.txt` and the terms of use; pause; introduce yourself; be careful with personal data. `robots.txt` is a request, not a lock.
- **BeautifulSoup** reads HTML (`select`, `get_text`); `find()` can return `None`. **Scrapy** handles big crawls through settings.
- This book only practises scraping on **self-made sites** on your own computer.

Next, in Chapter 42: interfaces. A script does not have to be used through a Python file: you will make commands in the terminal (CLI), a simple window (GUI), and see your first data dashboard.

Chapter 42 — Interfaces: Commands, Windows, and Dashboards

Series I5 — First Applications: Web, Maps, and Dashboards

In this chapter, you will:

- Build **command-line (CLI)** tools with `argparse` and **Typer**
- Test a CLI without opening a terminal (`subprocess` and `CliRunner`)
- Make a simple window (**GUI**) with `tkinter` and test it without showing it
- Build your first data **dashboard** as a self-contained HTML page
- Wrap the business-unit data in a CLI: the Series I5 running project

All scripts for this chapter are in the `code/ch42/` folder. On the web version, you can also [download them all at once \(zip\)](#).

The scripts you have written so far are run by typing `python name.py`. That is enough for you, but other people need a friendlier **way in**. Picture a food stall: some people order at the counter (fast, just say the order), some sit at a table with a picture menu, and some look at the notice board showing today's sales. One and the same kitchen, three ways to interact. Those are the three kinds of **interface** we discuss.

Three kinds of interface for the same program



Figure 0.1: Three kinds of interface for the same program.

In this chapter we use the same sample business-unit data (15 units in five sectors, **all numbers fictional**) from the file `data/business_units.csv`, read by the helper module `unit_data.py`.

CLI with argparse

A **CLI** (*Command-Line Interface*) is a program used by typing commands in a terminal, like `git commit -m "message"`. **argparse** is Python's built-in module for reading that typing.

```
# Chapter 42 - argparse: reading arguments from the command line
import argparse

parser = argparse.ArgumentParser(prog="feed_amount", description="Compute the daily
feed for a pond.")
```



```

parser.add_argument("biomass_kg", type=float, help="fish biomass (kg)") #
positional, required
parser.add_argument("--percent", type=float, default=3.0, help="feed percentage
(default 3.0)") # option
parser.add_argument("--detail", action="store_true", help="show the working") #
switch

# Normally argparse reads what you type in the terminal. Here we pass our own lists so
you can try it.
for typed in (["432.4"], ["432.4", "--percent", "2.5", "--detail"]):
    args = parser.parse_args(typed)
    feed = args.biomass_kg * args.percent / 100
    print(f"typed {typed} -> {args}")
    print(f"    feed: {feed:.2f} kg" + (" (working: biomass x percent / 100)" if
args.detail else ""))

print()
parser.print_help()

```

```

typed ['432.4'] -> Namespace(biomass_kg=432.4, percent=3.0, detail=False)
    feed: 12.97 kg
typed ['432.4', '--percent', '2.5', '--detail'] -> Namespace(biomass_kg=432.4,
percent=2.5, detail=True)
    feed: 10.81 kg (working: biomass x percent / 100)

usage: feed_amount [-h] [--percent PERCENT] [--detail] biomass_kg

Compute the daily feed for a pond.

positional arguments:
  biomass_kg          fish biomass (kg)

options:
  -h, --help          show this help message and exit
  --percent PERCENT   feed percentage (default 3.0)
  --detail            show the working

```

Three kinds of arguments appear here:

- **Positional** (`biomass_kg`): required, recognized by its order.
- **Option** (`--percent`): starts with a double hyphen, may be skipped, has a default value.
- **Switch** (`--detail, action="store_true"`): `True` when written, `False` when not.

`type=float` makes `argparse` turn the text "432.4" into a number. The result of `parse_args()` is a `Namespace` object whose attributes you read (`args.percent`). Normally `parse_args()` reads what is typed in the terminal; in this script we give it a **list** so it can be tried without a terminal (the same trick is used for testing). The arithmetic: $432.4 \times 3 / 100 = 12.97$ kg and $432.4 \times 2.5 / 100 = 10.81$ kg. The bottom of the output is the **automatic help** (`--help`) that `argparse` builds from the declarations, for free.

When the user mistypes

A good tool **rejects wrong input clearly**. We test the real `unit` tool (the helper module `cli_argparse.py`, shown below) through `subprocess` from Chapter 41:

```

# Chapter 42 - The "unit" command-line tool, argparse version (helper module; run
through scripts 02 and 03)
import argparse
import sys

from unit_data import read_units, summarize_by_sector

```

```

SECTORS = ["forest", "plantation", "paddy", "pond", "barn"]

def build_parser():
    parser = argparse.ArgumentParser(prog="unit", description="View the business-unit data (fictional).")
    sub = parser.add_subparsers(dest="command", required=True)

    list_cmd = sub.add_parser("list", help="show units")
    list_cmd.add_argument("--sector", choices=SECTORS, help="only one sector")
    list_cmd.add_argument("--max", type=int, default=0, help="row limit (0 = all)")

    sub.add_parser("summary", help="summary per sector")
    return parser

def main(argv=None):
    args = build_parser().parse_args(argv)
    units = read_units()
    if args.command == "list":
        if args.sector:
            units = [u for u in units if u["sector"] == args.sector]
        if args.max:
            units = units[: args.max]
        for u in units:
            print(f'{u["id"]:3} {u["sector"]:10} {u["unit"]:8} {u["value"]:8.1f} {u["uom"]}')
    else:
        for sector, r in summarize_by_sector(units).items():
            print(f'{sector:10} {r["count"]} units, mean {r["mean"]:.1f} {r["uom"]}')
    return 0

if __name__ == "__main__":
    sys.exit(main())

```

```

# Chapter 42 - argparse: what happens when the user mistypes
import subprocess
import sys

for arguments in (["list", "--sector", "lake"], ["list", "--max", "two"], []):
    result = subprocess.run(
        [sys.executable, "cli_argparse.py", *arguments],
        capture_output=True, text=True, encoding="utf-8",
    )
    print("arguments:", arguments)
    print("exit code:", result.returncode)
    print(result.stderr.strip().splitlines()[-1])
    print()

```

```

arguments: ['list', '--sector', 'lake']
exit code: 2
unit list: error: argument --sector: invalid choice: 'lake' (choose from 'forest', 'plantation', 'paddy', 'pond', 'barn')

arguments: ['list', '--max', 'two']
exit code: 2
unit list: error: argument --max: invalid int value: 'two'

arguments: []

```

```
exit code: 2
unit: error: the following arguments are required: command
```

The official documentation describes this behaviour: on invalid input, `parse_args()` prints a usage message to `stderr` and exits with **exit code 2**. The `choices=SECTORS` setting makes the sector `lake` rejected, `type=int` rejects `two`, and a subcommand with `required=True` rejects an empty input.

Subcommands

Big tools have several commands (`git commit`, `git push`). In `argparse`, that is done with `add_subparsers():list` and `summary` in our `unit` tool. Test it through a child program, exactly as people would use it:

```
# Chapter 42 - argparse: the "unit" tool with subcommands, tested through subprocess
import subprocess
import sys

def unit(*arguments):
    result = subprocess.run([sys.executable, "cli_argparse.py", *arguments],
                           capture_output=True, text=True, encoding="utf-8")
    return result.returncode, result.stdout

for arguments in (["summary"], ["list", "--sector", "pond"], ["list", "--max", "2"]):
    code, output = unit(*arguments)
    print(f"$ unit {' '.join(arguments)}    (exit code {code})")
    print(output)
```

```
$ unit summary    (exit code 0)
forest      3 units, mean 35.2 cm
plantation  4 units, mean 257.5 kg
paddy       2 units, mean 1115.2 kg
pond        3 units, mean 369.3 kg
barn        3 units, mean 383.8 kg

$ unit list --sector pond    (exit code 0)
L1 pond      Pond A      432.4 kg
L2 pond      Pond B      380.0 kg
L3 pond      Pond C      295.5 kg

$ unit list --max 2    (exit code 0)
H1 forest    Plot A      32.5 cm
H2 forest    Plot B      45.2 cm
```

CLI with Typer

`argparse` is strong, but wordy. **Typer** (a third-party library built on `Click`) turns it around: you write an **ordinary function with type hints** (Chapter 33), and `Typer` makes the command. A parameter without a default becomes a required argument; a parameter with a default becomes an option; a `bool` becomes a switch.

```
# Chapter 42 - Typer: an ordinary function becomes a command, tested without a terminal
import typer
from typer.testing import CliRunner

app = typer.Typer(add_completion=False)
```

```

@app.command()
def feed(biomass_kg: float, percent: float = 3.0, detail: bool = False):
    """Compute the daily feed for a pond."""
    result = biomass_kg * percent / 100
    typer.echo(f"feed: {result:.2f} kg")
    if detail:
        typer.echo("working: biomass x percent / 100")

runner = CliRunner()          # Typer's built-in tester: calls the app without opening
                              # a terminal

for typed in (["432.4"], ["432.4", "--percent", "2.5", "--detail"], ["lots"]):
    result = runner.invoke(app, typed)
    print(typed, "-> exit code", result.exit_code)
    print(result.output.strip().splitlines()[0] if result.exit_code == 0 else "(usage
error)")

result = runner.invoke(app, ["--help"])
print()
print(result.output)

```

```

['432.4'] -> exit code 0
feed: 12.97 kg
['432.4', '--percent', '2.5', '--detail'] -> exit code 0
feed: 10.81 kg
['lots'] -> exit code 2
(usage error)

```

Usage: feed [OPTIONS] {biomass_kg}

Compute the daily feed for a pond.

Arguments			
*	biomass_kg	<float>	[required]

Options			
--percent		<float>	[default: 3.0]
--detail	--no-detail		[default: no-detail]
--help			Show this message and exit.

- The three inputs give: success (12.97), success with an option (10.81), and a usage error (exit code 2) for lots, because it is not a number.
- CliRunner is **Typer's built-in tester**: it calls the app inside the same process and keeps the `exit_code` and output. No terminal needs to be opened, so it is fast and suits automatic tests.
- The automatic help uses a boxed look (made by the `rich` library). Notice `--detail` and `--no-detail`: Typer adds the switch pair by itself.

Here is the `unit` tool in its Typer version, with three commands (`list`, `summary`, `report`):

```

# Chapter 42 - The "unit" command-line tool, Typer version (helper module; used by
scripts 04 to 09)
from enum import Enum
from pathlib import Path
from typing import Annotated, Optional

import typer

```

```

from unit_data import read_units, summarize_by_sector

app = typer.Typer(help="View the business-unit data (fictional).",
no_args_is_help=True, add_completion=False)

class Sector(str, Enum):
    forest = "forest"
    plantation = "plantation"
    paddy = "paddy"
    pond = "pond"
    barn = "barn"

@app.command("list")
def list_units(
    sector: Annotated[Optional[Sector], typer.Option(help="only one sector")] = None,
    max_rows: Annotated[int, typer.Option("--max", min=0, help="row limit (0 = all)")]
= 0,
):
    """Show units."""
    units = read_units()
    if sector:
        units = [u for u in units if u["sector"] == sector.value]
    if max_rows:
        units = units[:max_rows]
    for u in units:
        typer.echo(f'{u["id"]:3} {u["sector"]:10} {u["unit"]:8} {u["value"]:8.1f}
{u["uom"]}')

@app.command()
def summary():
    """Summary per sector."""
    for sector, r in summarize_by_sector(read_units()).items():
        typer.echo(f'{sector:10} {r["count"]} units, mean {r["mean"]:8.1f} {r["uom"]}')

@app.command()
def report(output: Annotated[Path, typer.Argument(help="target Markdown file")]):
    """Write a short report to a file."""
    lines = ["# Business unit summary (fictional data)", ""]
    for sector, r in summarize_by_sector(read_units()).items():
        lines.append(f'- {sector}: {r["count"]} units, mean {r["mean"]:8.1f} {r["uom"]}')
    output.parent.mkdir(parents=True, exist_ok=True)
    output.write_text("\n".join(lines) + "\n", encoding="utf-8")
    typer.echo(f"report written: {output.name}")

if __name__ == "__main__":
    app()

```

An Enum (a fixed list of choices) replaces choices. Annotated[..., typer.Option(...)] adds a description and a limit (min=0). The functions above can be run as a program:

```

# Chapter 42 - Typer: the full "unit" tool run as a real program through subprocess
import subprocess
import sys

def unit(*arguments):

```

```

    result = subprocess.run([sys.executable, "cli_typer.py", *arguments],
                            capture_output=True, text=True, encoding="utf-8")
    return result.returncode, result.stdout + result.stderr

for arguments in (["--help"], ["list", "--sector", "barn"], ["list", "--sector",
"lake"]):
    code, text = unit(*arguments)
    print(f"$ unit {' '.join(arguments)}    (exit code {code})")
    print(text)

```

```
$ unit --help    (exit code 0)
```

```
Usage: cli_typer.py [OPTIONS] COMMAND [ARGS]...
```

```
View the business-unit data (fictional).
```

```

└─ Options ────────────────────────────────────────────────────────────────────
└─ --help          Show this message and exit.

└─ Commands ───────────────────────────────────────────────────────────────────
└─ list           Show units.
└─ summary        Summary per sector.
└─ report         Write a short report to a file.

```

```
$ unit list --sector barn    (exit code 0)
```

```
D1 barn      Pen 1      350.0 kg
```

```
D2 barn      Pen 2      412.5 kg
```

```
D3 barn      Pen 3      389.0 kg
```

```
$ unit list --sector lake    (exit code 2)
```

```
Usage: cli_typer.py list [OPTIONS]
```

```
Try 'cli_typer.py list --help' for help.
```

```

└─ Error ──────────────────────────────────────────────────────────────────────
└─ Invalid value for '--sector': 'lake' is not one of 'forest', 'plantation',
'paddy', 'pond', 'barn'.

```

The sector lake is rejected with exit code 2 and an error box naming the valid choices.

TIP

Choose **argparse** when you want zero dependencies (it is built into Python). Choose **Typer** when your tool has many commands and you like short code based on type hints. Both give automatic help and clear exit codes.

Testing a CLI

An untested CLI will break silently. With **CliRunner**, each behaviour becomes one check:

```

# Chapter 42 - Testing a Typer CLI with CliRunner (pytest style, written as a script)
from pathlib import Path

from typer.testing import CliRunner

from cli_typer import app

```

```

runner = CliRunner()
passed = 0

def check(condition, name):
    global passed
    print(("PASS " if condition else "FAIL ") + name)
    passed += bool(condition)

result = runner.invoke(app, ["summary"])
check(result.exit_code == 0 and "pond" in result.output, "summary shows the pond sector")

result = runner.invoke(app, ["list", "--sector", "paddy"])
check(len(result.output.strip().splitlines()) == 2, "list --sector paddy gives 2 rows")

result = runner.invoke(app, ["list", "--max", "4"])
check(len(result.output.strip().splitlines()) == 4, "list --max 4 gives 4 rows")

result = runner.invoke(app, ["list", "--sector", "lake"])
check(result.exit_code == 2, "unknown sector is rejected with exit code 2")

target = Path(__file__).parent / "_tmp" / "summary.md"
result = runner.invoke(app, ["report", str(target)])
check(result.exit_code == 0 and target.read_text(encoding="utf-8").startswith("# Business unit"),
       "report writes a Markdown file")

print(f"\n{passed} of 5 checks passed")

```

```

PASS summary shows the pond sector
PASS list --sector paddy gives 2 rows
PASS list --max 4 gives 4 rows
PASS unknown sector is rejected with exit code 2
PASS report writes a Markdown file

5 of 5 checks passed

```

This is the `pytest` style (Chapter 18) written as a script so the output shows. Notice that testing the **failure path** (an unknown sector must give code 2) is just as important as the success path. There are two ways to test a CLI in this chapter: `CliRunner` (fast, inside one process) and `subprocess` (a little slower, but exactly how people use it in a terminal).

A simple GUI with tkinter

A **GUI** (*Graphical User Interface*) is a program with windows: buttons, selection boxes, tables. **tkinter** is the GUI library that comes with Python (it wraps Tk). The official documentation notes that tkinter is an **optional** module: if it is missing from your copy of Python, look for instructions from whoever provided your Python. On the author's computer it is there (Tk version 9.0). **ttk** is the set of themed widgets recommended for new code.

```

# Chapter 42 - A tkinter window for viewing business units (helper module)
import tkinter as tk
from tkinter import ttk

from unit_data import read_units

SECTORS = ["all", "forest", "plantation", "paddy", "pond", "barn"]

```

```

def build_window(root):
    """Builds the window contents and returns the widgets that need testing."""
    root.title("Business units (fictional data)")
    choice = tk.StringVar(value="all")
    status = tk.StringVar()

    top = ttk.Frame(root, padding=8)
    top.pack(fill="x")
    ttk.Label(top, text="Sector:").pack(side="left")
    box = ttk.Combobox(top, textvariable=choice, values=SECTORS, state="readonly",
width=10)
    box.pack(side="left", padx=6)

    table = ttk.Treeview(root, columns=("unit", "kind", "value"), show="headings",
height=8)
    for column, title in (("unit", "Unit"), ("kind", "Kind"), ("value", "Value")):
        table.heading(column, text=title)
    table.pack(fill="both", expand=True, padx=8)
    ttk.Label(root, textvariable=status, padding=8).pack(anchor="w")

    def load(*_):
        table.delete(*table.get_children())
        chosen = [u for u in read_units() if choice.get() in ("all", u["sector"])]
        for u in chosen:
            table.insert("", "end", values=(u["unit"], u["kind"], f'{u["value"]}'
{u["uom"]}))
        status.set(f"{len(chosen)} units shown")

    box.bind("<<ComboboxSelected>>", load)
    load()
    return choice, table, status, load

if __name__ == "__main__":
    window = tk.Tk()
    build_window(window)
    window.mainloop()           # shows the window and waits for the user

```

This window has a sector selection box, a table of units, and a status line. Choosing a sector calls `load()`, which empties the table and fills it again. Notice the good pattern: the whole window content is built by the function `build_window(root)`, while `mainloop()` lives **only** in the `if __name__ == "__main__":` part.

This code was **not run** when the book was built (the `mainloop()` part) because a visible window needs a screen and a user, so no output is shown. You run it yourself with `python gui_unit.py`. But its content can still be tested without showing anything, because the structure is separated:

```

# Chapter 42 - tkinter: testing the window WITHOUT showing it (the window is hidden)
import tkinter as tk

from gui_unit import build_window

root = tk.Tk()
root.withdraw()           # hide it: no window appears on the screen
choice, table, status, load = build_window(root)
root.update()

print("window title :", root.title())
print("at the start :", status.get(), "| table rows:", len(table.get_children()))

for sector in ("pond", "forest"):
    choice.set(sector)      # as if the user picked it in the combo box

```



```

load()
rows = [table.item(i, "values")[0] for i in table.get_children()]
print(f"{sector:7}      :", status.get(), "->", rows)

root.destroy()

```

```

window title : Business units (fictional data)
at the start : 15 units shown | table rows: 15
pond          : 3 units shown -> ['Pond A', 'Pond B', 'Pond C']
forest        : 3 units shown -> ['Plot A', 'Plot B', 'Plot C']

```

`root.withdraw()` hides the window so nothing appears on screen. Then we **imitate the user**: `choice.set("pond")` is like picking in the box, and `load()` is like the selection event. The results match: 15 units at the start, 3 pond units (Pond A, B, C), and 3 forest units (Plot A, B, C).

i REMEMBER

What is tested here is the window's **logic** (the table contents, the status text). The looks (button positions, colours, sizes) cannot be checked without seeing the window itself. Checking the looks is still your job, with your own eyes, on your computer.

An introductory data dashboard

A **dashboard** is a single page that summarizes the important numbers with tables and charts, like a car's instrument panel. The simplest dashboard needs no server: one self-contained HTML file that you can open with a double click.

```

# Chapter 42 - Introductory dashboard: one self-contained HTML page with a table and
a chart
import base64
import io
from pathlib import Path

import matplotlib.pyplot as plt
import pandas as pd

from unit_data import FILE

df = pd.read_csv(FILE)
summary = df.groupby("sector", sort=False)["value"].agg(["count", "mean"]).round(1)
print(summary)

fig, ax = plt.subplots(figsize=(6, 3))
summary["count"].plot.bar(ax=ax, color="#4f8a3b")
ax.set_ylabel("number of units")
ax.set_xlabel("")
fig.tight_layout()
buf = io.BytesIO()
fig.savefig(buf, format="png", dpi=100)
plt.close(fig)
picture = base64.b64encode(buf.getvalue()).decode("ascii")

page = f"""<!doctype html>
<html lang="en"><head><meta charset="utf-8"><title>Business unit dashboard</title></
head>
<body style="font-family: sans-serif; max-width: 640px; margin: 2em auto">
<h1>Business unit dashboard (fictional data)</h1>
<p>Total units: <b>{len(df)}</b> in <b>{df['sector'].nunique()}</b> sectors.</p>


```

```
{summary.to_html(classes="summary")}}
</body></html>
"""
target = Path(__file__).parent / "_tmp" / "dashboard.html"
target.parent.mkdir(exist_ok=True)
target.write_text(page, encoding="utf-8")
print("\ndashboard.html created:", target.stat().st_size > 5000, "(open it with a double click)")
print("has the embedded chart :", "data:image/png;base64," in page)
```

	count	mean
sector		
forest	3	35.2
plantation	4	257.5
paddy	2	1115.2
pond	3	369.3
barn	3	383.8

```
dashboard.html created: True (open it with a double click)
has the embedded chart : True
```

The chart is made with matplotlib, turned into **base64** text, then embedded directly in the HTML (`data:image/png;base64,...`), so the file needs no separate image. The summary table comes from pandas' `to_html()`. We check that the file was made and contains the chart; we **cannot** see how it looks from this script, so open `_tmp/dashboard.html` yourself if you want to see it.

The weakness of a static dashboard: its numbers are **frozen** when the file is made and there is no interaction. A live dashboard that can be clicked around and shows a map is covered in Chapter 43.

Running Project I5: a CLI for business units

In Chapter 41 the weekly report was fetched by scraping. Now we hand colleagues the `unit` tool: they only have to type a command. The daily automation script below calls that CLI like a person at a terminal, checks the exit code of each step, and stops when one fails.

```
# Chapter 42 - Running project I5: the "unit" CLI called by a daily automation script
import subprocess
import sys
from pathlib import Path

target = Path(__file__).parent / "_tmp" / "daily" / "summary.md"

def unit(*arguments):
    result = subprocess.run([sys.executable, "cli_typer.py", *arguments],
                           capture_output=True, text=True, encoding="utf-8")
    return result.returncode, result.stdout.strip()

steps = [
    ("summary", ["summary"]),
    ("first two ponds", ["list", "--sector", "pond", "--max", "2"]),
    ("write report", ["report", str(target)]),
]
for name, arguments in steps:
    code, output = unit(*arguments)
    print(f"[{name}] exit code {code}")
    print(output)
    if code != 0:
        # good automation stops when one step fails
        raise SystemExit(f"step '{name}' failed")
```

```
print("\ncontents of the report file:")
print(target.read_text(encoding="utf-8"))
```

```
[summary] exit code 0
forest      3 units, mean 35.2 cm
plantation  4 units, mean 257.5 kg
paddy       2 units, mean 1115.2 kg
pond        3 units, mean 369.3 kg
barn        3 units, mean 383.8 kg
[first two ponds] exit code 0
L1 pond      Pond A      432.4 kg
L2 pond      Pond B      380.0 kg
[write report] exit code 0
report written: summary.md

contents of the report file:
# Business unit summary (fictional data)

- forest: 3 units, mean 35.2 cm
- plantation: 4 units, mean 257.5 kg
- paddy: 2 units, mean 1115.2 kg
- pond: 3 units, mean 369.3 kg
- barn: 3 units, mean 383.8 kg
```

Three steps ran, all with exit code 0. The report file holds the mean of each sector. Take plantation: 4 units with a mean of $(245.5 + 310.0 + 198.5 + 276.0) / 4 = 1030.0 / 4 = 257.5$ kg. The habit of `raise SystemExit(...)` when a step fails keeps the automation from carrying on with work built on a broken result.

Exercises

1. Add a `--tall` option (a switch) to `01_argparse_basics.py`; when written, also print “feed per fish” for 800 fish.
2. In `cli typer.py`, add a command `one` that takes a `unit_id` (e.g. D2) and prints that unit, or “not found” with exit code 1. Add two checks in `06_cli_tests.py`.
3. Change `gui_unit.py` so the status also shows the highest value currently displayed, and update `07_tkinter_test.py` to check it.
4. **Challenge:** give the `unit` CLI a `dashboard` command that calls the code of `08_static_dashboard.py` and prints the location of the file. Test it with `CliRunner`.

Chapter Summary

- There are three **interfaces**: command line (CLI), window (GUI), and web page (dashboard).
- **argparse** (built in): positional arguments, options, switches, **choices**, subcommands. Wrong input: a message to `stderr` and exit code 2.
- **Typer**: a function with type hints becomes a command; test with `CliRunner (exit_code, output)`.
- Test a CLI on two paths: success **and** failure; `CliRunner` or `subprocess`.
- **tkinter**: separate building the window from `mainloop()` so the logic can be tested with `withdraw()`; the looks still need a human.
- A static **dashboard** = one self-contained HTML with an embedded chart; the interactive dashboard waits in Chapter 43.

Next, in Chapter 43: quick data apps and live dashboards: Streamlit, Gradio, Dash, interactive maps, HTMX, how to serve them to users, and the closing of Series I5.

Chapter 43 — Quick Data Apps and Dashboards: From Script to Someone Else's Screen

Series I5 — First Applications: Web, Maps, and Dashboards

In this chapter, you will:

- Build a dashboard with **Streamlit** and test it without a browser (**AppTest**)
- Build a function demo with **Gradio** and a custom dashboard with **Dash**, and test them
- Show an **interactive map** (folium and pydeck) in a dashboard
- Meet **HTMX** for lightweight web pages and **FastAPI** for data APIs
- Understand how to **serve** an application to users, then close Series I5

All scripts for this chapter are in the `code/ch43/` folder. On the web version, you can also [download them all at once \(zip\)](#).

Chapter 42 ended with a static dashboard: nice, but frozen. This time we build a **live** dashboard: the user picks a sector, the numbers change, the map shifts. Luckily Python has libraries that make this possible without learning JavaScript. Picture a shop: before, you stuck a price list on the wall (the static dashboard); now you have a cashier who answers customers' questions one by one.

Choosing a tool for a data application

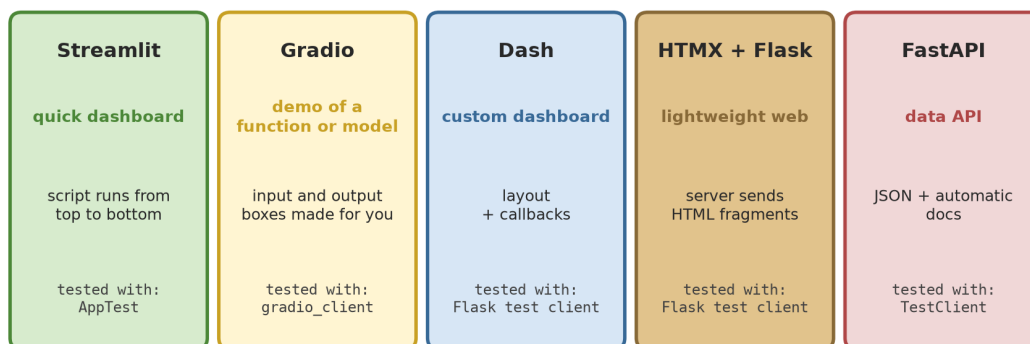


Figure 0.1: Choosing a tool for a data application.

We use the same sample business-unit data as the previous chapters (15 units, five sectors, **all numbers fictional**). All libraries below are installed on the author's computer and were **tested by running them**; library versions keep changing, so if behaviour differs on your computer, check the latest official documentation.

One honest limitation: this chapter **cannot show how things look** in a browser. Everything run here is tested through code (without opening a browser), so what is proven is the **logic and the content**, not the looks.

Streamlit: a dashboard from one script

Streamlit turns an ordinary Python script into a web page. The idea is simple: the script is read **from top to bottom**, and every time the user changes something (picks, slides), the **whole script is run again**. No HTML, no JavaScript.

```
# Chapter 43 - Streamlit dashboard per sector (helper module; run with: streamlit
run app_streamlit.py)
import pandas as pd
import streamlit as st

from map_view import build_map
from unit_data import read_units

SECTORS = ["all", "forest", "plantation", "paddy", "pond", "barn"]

@st.cache_data                                # the result is kept; the CSV is not re-read on
every interaction
def load_units():
    return pd.DataFrame(read_units())

st.set_page_config(page_title="Business unit dashboard", layout="wide")
st.title("Business unit dashboard (fictional data)")

sector = st.sidebar.selectbox("Sector", SECTORS)
units = load_units()
shown = units if sector == "all" else units[units["sector"] == sector]

left, middle, right = st.columns(3)
left.metric("Number of units", len(shown))
middle.metric("Mean value", f"{shown['value'].mean():.1f}")
right.metric("Highest value", f"{shown['value'].max():.1f}")

if sector == "all":
    st.info("Each sector has a different unit of measure (cm or kg): pick one sector
to compare.")
else:
    st.bar_chart(shown.set_index("unit")["value"])

st.dataframe(shown[["id", "sector", "unit", "kind", "value", "uom"]], hide_index=True)
st.map(shown, latitude="lat", longitude="lon")

with st.expander("Interactive map (folium)":
    st.iframe(build_map(shown.to_dict("records")).get_root().render(), height=420)
```

Read it like a recipe:

- `st.sidebar.selectbox(...)` makes a selection box in the sidebar and **returns** the current choice.
- `st.columns(3)` and `.metric(...)` make three number cards.
- `st.bar_chart`, `st.dataframe`, and `st.map` make a chart, a table, and a point map. `st.map` is given the names of the latitude and longitude columns (lat and lon).
- `@st.cache_data` keeps the result of `load_units()` so the CSV is not re-read on every interaction.
- The last part embeds a **folium map** in a collapsible section (covered in the map part).

The app is started with `streamlit run app_streamlit.py`, which opens an address on your computer. **That command is not run when the book is built**, because it starts a server that waits for a browser. Instead, Streamlit provides **AppTest** (official documentation: testing without a browser): it runs the script in memory and gives access to its elements.

```
# Chapter 43 - Streamlit: running the dashboard WITHOUT a browser using AppTest
import logging

from streamlit.testing.v1 import AppTest

# "bare mode": AppTest has no browser, so Streamlit grumbles quietly; silence that
warning
logging.getLogger("streamlit.runtime.scriptrunner_utils.script_run_context").setLevel(logging.ERROR)

at = AppTest.from_file("app_streamlit.py", default_timeout=30).run()

print("has an error      :", bool(at.exception))
print("title            :", at.title[0].value)
print("sector choices   :", at.selectbox[0].options)
print("chosen sector    :", at.selectbox[0].value)
for m in at.metric:
    print(f"metric          : {m.label} = {m.value}")
print("table rows       :", len(at.dataframe[0].value))
print("info box         :", at.info[0].value[:40] + "...")
```

```
has an error      : False
title            : Business unit dashboard (fictional data)
sector choices   : ['all', 'forest', 'plantation', 'paddy', 'pond', 'barn']
chosen sector    : all
metric          : Number of units = 15
metric          : Mean value = 375.0
metric          : Highest value = 1250.0
table rows       : 15
info box         : Each sector has a different unit of meas...
```

At the start, the chosen sector is all, and the numbers are right: 15 units, a mean value of 375.0 (this number mixes cm and kg, which is why the app shows an info box warning about it), and a highest value of 1250.0 (paddy field 1). Now we imitate a user:

```
# Chapter 43 - Streamlit: imitating a user choosing a sector, then checking the result
import logging

from streamlit.testing.v1 import AppTest

logging.getLogger("streamlit.runtime.scriptrunner_utils.script_run_context").setLevel(logging.ERROR)

at = AppTest.from_file("app_streamlit.py", default_timeout=30).run()

for sector in ["forest", "pond", "barn"]:
    at.selectbox[0].select(sector).run()          # like a user clicking a choice in
the sidebar
    values = {m.label: m.value for m in at.metric}
    chart = len(at.get("vega_lite_chart"))        # st.bar_chart = a Vega-Lite
chart element
    the_map = len(at.get("deck_gl_json_chart"))    # st.map = a deck.gl map element
    print(f"{sector:10} units={values['Number of units']} mean={values['Mean value']} "
          f"highest={values['Highest value']} chart={chart} map={the_map}")
    rows={len(at.dataframe[0].value)}")

# Choosing something that is not in the list fails when the app is re-run
try:
    at.selectbox[0].select("lake").run()
except ValueError as e:
    print("invalid choice rejected:", type(e).__name__)
```

```
forest    units=3 mean=35.2 highest=45.2 chart=1 map=1 rows=3
pond      units=3 mean=369.3 highest=432.4 chart=1 map=1 rows=3
barn      units=3 mean=383.8 highest=412.5 chart=1 map=1 rows=3
invalid choice rejected: ValueError
```

`at.selectbox[0].select("pond").run()` is “the user picks pond, the script runs again”. The result matches a hand calculation: pond has 3 units, a mean of $(432.4 + 380.0 + 295.5) / 3 = \mathbf{369.3}$, and a highest value of **432.4**. A bar chart and a map each appear as one element. Choosing something that is not in the list (lake) is answered by a `ValueError` when the app is re-run.

i REMEMBER

AppTest proves that the app’s logic is right and that no exception occurs. It does **not** prove that the chart looks nice. Charts and maps only show up as elements of certain types (`vega_lite_chart`, `deck_gl_json_chart`); the shape on screen still has to be looked at by a human.

Gradio: a demo of a function or model

Gradio has a slightly different purpose from Streamlit: wrapping a **function** with automatic input and output boxes. It is much loved for showing off a **model**: the user types something, the model answers. Here our “model” is deliberately simple: a formula estimating a fish’s weight from its length, a **fictional example**, not research.

```
# Chapter 43 - Gradio demo: find units and a "model" that estimates fish weight
(helper module)
import gradio as gr

from unit_data import read_units

SECTORS = ["all", "forest", "plantation", "paddy", "pond", "barn"]

def find_units(sector, minimum_value):
    """Returns a table of units and a summary sentence."""
    chosen = [
        [u["id"], u["sector"], u["unit"], u["value"], u["uom"]]
        for u in read_units()
        if sector in ("all", u["sector"]) and u["value"] >= minimum_value
    ]
    return chosen, f"{len(chosen)} matching units"

def estimate_weight(length_cm):
    """Stand-in for a 'model': weight = 0.01 x length^3 (a FICTIONAL example formula,
    not research)."""
    return round(0.01 * length_cm ** 3, 1)

find_interface = gr.Interface(
    fn=find_units,
    inputs=[gr.Dropdown(SECTORS, value="all", label="Sector"),
            gr.Slider(0, 1000, value=0, step=10, label="Minimum value")],
    outputs=[gr.Dataframe(headers=["id", "sector", "unit", "value", "uom"],
                           label="Units"),
            gr.Textbox(label="Summary")],
    api_name="find",
    flagging_mode="never",
)
estimate_interface = gr.Interface(
```

```

    fn=estimate_weight,
    inputs=gr.Number(value=25, label="Fish length (cm)",
    outputs=gr.Number(label="Estimated weight (g)",
    api_name="estimate",
    flagging_mode="never",
    )
demo = gr.TabbedInterface([find_interface, estimate_interface], ["Find units",
"Estimate weight"],
                        title="Business unit demo (fictional data)")

if __name__ == "__main__":
    demo.launch() # local address only; share=True is NOT used

```

Two functions, two `gr.Interface` objects, combined into tabs. `api_name` names each access point so other programs can call it. There are two ways to test:

```

# Chapter 43 - Gradio: test the functions directly, then start the app on your
own computer
import contextlib
import io
import os

os.environ["GRADIO_ANALYTICS_ENABLED"] = "False" # do not send usage statistics
os.environ["HF_HUB_OFFLINE"] = "1" # do not contact huggingface.co

from gradio_client import Client

from app_gradio import demo, estimate_weight, find_units

# 1. Ordinary functions: no server is needed to test them
table, summary = find_units("pond", 300)
print("direct      :", summary, "->", [row[2] for row in table])
print("direct      :", estimate_weight(25), "g for a 25 cm fish")

# 2. Start a local server, then call it through the official client (like a user over
the network)
demo.launch(server_name="127.0.0.1", prevent_thread_lock=True, share=False,
quiet=True)
try:
    client = Client(demo.local_url, verbose=False)
    data, text = client.predict("barn", 0, api_name="/find")
    print("client find :", text, "->", [r[2] for r in data["data"]])
    print("client est. :", client.predict(30, api_name="/estimate"), "g for a 30
cm fish")
finally:
    with contextlib.redirect_stdout(io.StringIO()): # hides the port-number message
        demo.close()
print("server closed")

```

```

direct      : 2 matching units -> ['Pond A', 'Pond B']
direct      : 156.2 g for a 25 cm fish
client find : 3 matching units -> ['Pen 1', 'Pen 2', 'Pen 3']
client est. : 270.0 g for a 30 cm fish
server closed

```

1. **Call the function directly:** 2 pond units with value ≥ 300 (Pond A and B), and $0.01 \times 25^3 = 156.25$, which Python rounds to 156.2 g.
2. **Start a local server and call it through `gradio_client`,** the official client: like a remote user. 3 barn units match, and for a 30 cm fish: $0.01 \times 30^3 = 270.0$ g.

Three notes on safety and privacy:

- We **do not use** `share=True`. According to the Gradio documentation, that option creates a **temporary public link** (valid for about a week) that forwards traffic to your computer; anyone with the link can use the app, while your computer does the work. Do not do that with data that is not meant for the public.
- While testing, we monitored network connections and found that **Gradio contacts huggingface.co** unless prevented. We did not investigate its purpose. With `HF_HUB_OFFLINE=1` (and `GRADIO_ANALYTICS_ENABLED=False` for usage statistics), no connection leaving this computer was detected any more. If you use Gradio for sensitive data, check these settings.
- `demo.launch()` without arguments only serves your own computer.

Dash: a dashboard you control completely

Dash (from Plotly) works more “from the bottom”: you assemble a **layout** (the page skeleton made of components) and **callbacks** (functions called when a component changes). More code than Streamlit, but finer control.

```
# Chapter 43 - Dash dashboard (helper module; run with: python app_dash.py)
import pandas as pd
import plotly.express as px
from dash import Dash, Input, Output, dcc, html

from unit_data import read_units

SECTORS = ["forest", "plantation", "paddy", "pond", "barn"]
units = pd.DataFrame(read_units())

app = Dash(__name__)
app.layout = html.Div([
    html.H1("Dash dashboard (fictional data)"),
    dcc.Dropdown(id="sector", options=SECTORS, value="pond", clearable=False),
    html.Div(id="summary"),
    dcc.Graph(id="chart"),
])

@app.callback(Output("chart", "figure"), Output("summary", "children"),
              Input("sector", "value"))
def update(sector):
    part = units[units["sector"] == sector]
    figure = px.bar(part, x="unit", y="value", title=f"Sector {sector}")
    return figure, f"{len(part)} units, mean {part['value'].mean():.1f}"
    {part['uom'].iloc[0]}"

if __name__ == "__main__":
    app.run(debug=False) # http://127.0.0.1:8050
```

The `@app.callback` decorator says: “*when the value of `sector` changes, call this function and put the results in `chart.figure` and `summary.children`.*” Dash stands on Flask, so we can use the **Flask test client** without a browser:

```
# Chapter 43 - Dash: testing the dashboard through Flask's test client, without a browser
from app_dash import app, update

client = app.server.test_client() # Dash is built on top of Flask

page = client.get("/")
print("home page :", page.status_code, "| loads the Dash renderer:", b"_dash-renderer"
      in page.data)

layout = client.get("/_dash-layout").get_json()
```

```

children = layout["props"]["children"]
print("layout      :", [(c["type"], c["props"].get("id")) for c in children])

# Imitate the request the browser sends when the user changes the sector
body = {
    "output": "..chart.figure...summary.children..",
    "outputs": [{ "id": "chart", "property": "figure"}, { "id": "summary", "property":
"children"}],
    "inputs": [{ "id": "sector", "property": "value", "value": "paddy"}],
    "changedPropIds": ["sector.value"],
}
reply = client.post("/_dash-update-component", json=body)
print("callback    :", reply.status_code, reply.get_json()["response"]["summary"]
["children"])

# The function can also be called directly, like any ordinary function
print("direct      :", update("barn")[1])

```

```

home page : 200 | loads the Dash renderer: True
layout    : [('H1', None), ('Dropdown', 'sector'), ('Div', 'summary'), ('Graph',
'chart')]
callback  : 200 2 units, mean 1115.2 kg
direct    : 3 units, mean 383.8 kg

```

The home page answers 200 and loads the Dash renderer. The layout holds the `H1` heading, the `sector` dropdown, the `summary`, and the `chart`. We then send the same request the browser sends when the user picks **paddy**: the answer is “2 units, mean 1115.2 kg”, matching $(1250.0 + 980.5) / 2 = 1115.25$ rounded to one decimal. The callback function can also be called directly.

TIP

A rule of thumb: **Streamlit** for quick dashboards and prototypes; **Gradio** to show off one function or model; **Dash** when layout and interaction need detailed control. For a simple internal dashboard, start with Streamlit.

Interactive maps in a dashboard

In Chapters 38 to 40 you made maps. For maps in a dashboard, **folium** wraps the JavaScript map library **Leaflet**; **pydeck** wraps **deck.gl** for point layers or big data. Both produce HTML.

```

# Chapter 43 - Helper module: an interactive map of the unit locations with folium
(FICTIONAL coordinates)
import folium

COLORS = {"forest": "darkgreen", "plantation": "orange", "paddy": "green", "pond":
"blue", "barn": "red"}

def build_map(units):
    """One marker per unit, coloured by sector, with details when clicked."""
    center = [sum(u["lat"] for u in units) / len(units), sum(u["lon"] for u in
units) / len(units)]
    the_map = folium.Map(location=center, zoom_start=13, tiles="OpenStreetMap")
    for u in units:
        folium.Marker(
            [u["lat"], u["lon"]],
            tooltip=f'{u["unit"]} ({u["sector"]})',
            popup=f'{u["unit"]}: {u["value"]} {u["uom"]}',
            icon=folium.Icon(color=COLORS[u["sector"]]),

```

```

    ).add_to(the_map)
    return the_map

```

```

# Chapter 43 - Interactive maps: folium (Leaflet) and pydeck (deck.gl), saved as
HTML files
from pathlib import Path

import pydeck as pdk

from map_view import build_map
from unit_data import read_units

units = read_units()
out = Path(__file__).parent / "_tmp"
out.mkdir(exist_ok=True)

# 1. folium: one marker per unit
the_map = build_map(units)
file = out / "folium_map.html"
the_map.save(file)
html = file.read_text(encoding="utf-8")
print("folium : markers =", html.count("L.marker("), "| mentions Leaflet:", "leaflet"
in html.lower())
print("      HTML file created:", file.stat().st_size > 5000)

# 2. pydeck: a point layer sized by value (only fair for sectors that share a unit
of measure)
ponds = [u for u in units if u["sector"] == "pond"]
layer = pdk.Layer("ScatterplotLayer", data=ponds, get_position=["lon", "lat"],
                  get_radius="value", radius_scale=0.5, get_fill_color=[30, 100,
200, 160])
view = pdk.ViewState(latitude=ponds[0]["lat"], longitude=ponds[0]["lon"], zoom=13)
deck = pdk.Deck(layers=[layer], initial_view_state=view)
file2 = out / "pydeck_map.html"
deck.to_html(str(file2), open_browser=False)
print("pydeck : pond points =", len(ponds), "| HTML file created:", file2.stat().st_size
> 1000)

# Both files load their map library from the internet when OPENED in a browser (not
when made here)
print("libraries loaded from a CDN:", "https://" in html and "https://" in
file2.read_text(encoding="utf-8"))

```

```

folium : markers = 15 | mentions Leaflet: True
      HTML file created: True
pydeck : pond points = 3 | HTML file created: True
libraries loaded from a CDN: True

```

There are 15 markers (one per unit) on the folium map, and 3 pond points on the pydeck map. Notice the last line: both files **load their map libraries from the internet when opened** in a browser, not when made. So these maps need an internet connection to be viewed; we did not open them here. Streamlit embeds the folium map with `st.iframe(...)`. (The older function `st.components.v1.html` is marked deprecated in Streamlit 1.65, with `st.iframe` as its replacement.) The Streamlit documentation also warns: **do not embed HTML from untrusted sources** this way; our map is made by ourselves from our own data.

⚠ WARNING

The point size on the pydeck map uses the `value` column. That is **fair only for sectors with the same unit of measure**. We chose the pond sector (all in kg). Drawing forest (cm) next to pond (kg) on one scale would mislead.

HTMX: lightweight web without a big framework

Sometimes all you need is a page where part of the content changes without reloading the whole page. **HTMX** is a small JavaScript library that gives HTML that ability through **attributes**: `hx-get` (ask this address), `hx-trigger` (when), `hx-target` (fill which part), `hx-swap` (how to replace). The server, **Flask** here, only has to send an **HTML fragment**, not JSON.

```
# Chapter 43 - Lightweight web with HTMX + Flask (helper module; run with: flask --app
app_htmx run)
from flask import Flask, abort, render_template_string, request

from unit_data import read_units

app = Flask(__name__)
SECTORS = ["forest", "plantation", "paddy", "pond", "barn"]

PAGE = """<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8"><title>Business units (HTMX)</title>
  <script src="https://cdn.jsdelivr.net/npm/htmx.org@2.0.11/dist/htmx.min.js"></
script>
</head>
<body>
  <h1>Business units (fictional data)</h1>
  <select name="s" hx-get="/sector" hx-trigger="change" hx-target="#result" hx-
swap="innerHTML">
    <option value="">- choose a sector -</option>
    {% for s in sectors %}<option value="{{ s }}">{{ s }}</option>{% endfor %}
  </select>
  <div id="result">Choose a sector above.</div>
</body>
</html>"""

TABLE = """<table>
  {% for u in units %}<tr><td>{{ u.unit }}</td><td>{{ u.kind }}</td><td>{{ u.value }}
{{ u.uom }}</td></tr>
  {% endfor %}
</table>"""

@app.get("/")
def index():
    return render_template_string(PAGE, sectors=SECTORS)

@app.get("/sector")
def per_sector():
    s = request.args.get("s", "")
    if s not in SECTORS:
        abort(400, "unknown sector")
    units = [u for u in read_units() if u["sector"] == s]
```

```

    return render_template_string(TABLE, units=units)        # only an HTML fragment,
not a whole page

```

Choosing a sector sends GET `/sector?s=pond`, and the server answers a small table that is put into `#result`. The HTMX script is loaded from the jsDelivr CDN; version 2.0.11 is the latest version in the npm registry at the time of writing (read on 2026-10-10). In a real deployment you can add the `integrity` attribute (*Subresource Integrity*) as in the HTMX documentation's example; we did not copy that hash from memory.

```

# Chapter 43 - HTMX + Flask: the server side tested with the test client (htmx JavaScript
is NOT run here)
from app_htmx import app

client = app.test_client()

page = client.get("/").get_data(as_text=True)
print("home page:", len(page.splitlines()), "lines")
for attribute in ("hx-get=", "hx-trigger=", "hx-target=", "htmx.org@2.0.11"):
    print(f"  contains {attribute:18}", attribute in page)

# What htmx sends when the user chooses: GET /sector?s=pond with the HX-Request header
reply = client.get("/sector?s=pond", headers={"HX-Request": "true"})
fragment = reply.get_data(as_text=True)
print("\nGET /sector?s=pond ->", reply.status_code, "| HTML fragment, not a full page:",
"<html" not in fragment)
print(fragment)

print("odd sector ->", client.get("/sector?s=lake").status_code)
print("no sector ->", client.get("/sector").status_code)

```

```

home page: 15 lines
  contains hx-get=           True
  contains hx-trigger=       True
  contains hx-target=       True
  contains htmx.org@2.0.11   True

GET /sector?s=pond -> 200 | HTML fragment, not a full page: True
<table>
  <tr><td>Pond A</td><td>tilapia</td><td>432.4 kg</td></tr>
  <tr><td>Pond B</td><td>catfish</td><td>380.0 kg</td></tr>
  <tr><td>Pond C</td><td>tilapia</td><td>295.5 kg</td></tr>

</table>
odd sector -> 400
no sector -> 400

```

What is proven: the page contains the HTMX attributes and the version 2.0.11 script; the `/sector` endpoint answers an HTML fragment (not a full page) holding the three ponds; an odd or missing sector is rejected with code 400. **What is not tested:** HTMX's JavaScript in a browser, that is, that choosing a sector really replaces the content of `#result`. That needs a browser and we did not open one.

FastAPI: a door for other programs

A dashboard is a door for **people**. An **API** is a door for **other programs**: a phone app, another script, or a dashboard elsewhere asking for data as JSON. In Chapter 36 you met FastAPI; now we use it for the data of all business units, with the Pydantic models of Chapter 34.

```

# Chapter 43 - FastAPI API for the data of all business units (helper module)
from typing import Literal

```

```

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

from unit_data import read_units, summarize_by_sector

Sector = Literal["forest", "plantation", "paddy", "pond", "barn"]

class Unit(BaseModel):
    id: str
    sector: Sector
    unit: str
    kind: str
    lat: float
    lon: float
    value: float
    uom: str

class Summary(BaseModel):
    sector: Sector
    count: int
    total: float
    mean: float
    uom: str

app = FastAPI(title="Business unit API (fictional data)")

@app.get("/unit", response_model=list[Unit])
def list_units(sector: Sector | None = None):
    return [u for u in read_units() if sector is None or u["sector"] == sector]

@app.get("/unit/{unit_id}", response_model=Unit)
def one_unit(unit_id: str):
    for u in read_units():
        if u["id"] == unit_id:
            return u
    raise HTTPException(status_code=404, detail=f"unit {unit_id} does not exist")

@app.get("/summary", response_model=list[Summary])
def summary():
    return [{"sector": s, **r} for s, r in summarize_by_sector(read_units()).items()]

# Chapter 43 - FastAPI: an API for the data of all business units, tested with TestClient
from fastapi.testclient import TestClient

from api_unit import app

client = TestClient(app)

everything = client.get("/unit").json()
print("GET /unit      :", len(everything), "units; first =", everything[0]
      ["id"], everything[0]["unit"])

ponds = client.get("/unit", params={"sector": "pond"}).json()
print("GET /unit?sector=pond :", [u["unit"] for u in ponds])

```

```

one = client.get("/unit/D2")
print("GET /unit/D2          :", one.status_code, one.json()["value"], one.json()["uom"])

missing = client.get("/unit/X9")
print("GET /unit/X9          :", missing.status_code, missing.json())

wrong = client.get("/unit", params={"sector": "lake"})
print("GET /unit?sector=lake :", wrong.status_code, wrong.json()["detail"][0]["type"])

print("GET /summary          :", [(r["sector"], r["count"], round(r["mean"], 1)) for r in client.get("/summary").json()])
print("automatic docs       :", client.get("/docs").status_code, "| schema:", client.get("/openapi.json").status_code)

```

```

GET /unit          : 15 units; first = H1 Plot A
GET /unit?sector=pond : ['Pond A', 'Pond B', 'Pond C']
GET /unit/D2        : 200 412.5 kg
GET /unit/X9        : 404 {'detail': 'unit X9 does not exist'}
GET /unit?sector=lake : 422 literal_error
GET /summary        : [('forest', 3, 35.2), ('plantation', 4, 257.5), ('paddy', 2, 1115.2), ('pond', 3, 369.3), ('barn', 3, 383.8)]
automatic docs      : 200 | schema: 200

```

TestClient calls the API without starting a server. The result shows: 15 units; three ponds when filtered; D2 found (412.5 kg); X9 gives **404** with a clear message; the sector `lake` gives **422**, an automatic validation error because it is not one of the `Literal` values; and FastAPI builds interactive documentation (`/docs`) and a schema (`/openapi.json`) without you writing them.

Serving an application to users

So far everything has run on `127.0.0.1`, an address that **this computer only** can reach. For other people to use it, the application must run on a machine they **can reach**. Let us look at the first real server, **uvicorn**, which runs our API over genuine HTTP:

```

# Chapter 43 - Serving an application: a real server (uvicorn) on your own computer +
serving files
import socket
import threading
import time
from pathlib import Path

import httpx
import uvicorn

from api_unit import app

# Find a free port, then start the server in a separate thread
with socket.socket() as s:
    s.bind(("127.0.0.1", 0))
    port = s.getsockname()[1]

server = uvicorn.Server(uvicorn.Config(app, host="127.0.0.1", port=port,
log_level="warning"))
thread = threading.Thread(target=server.run, daemon=True)
thread.start()
while not server.started:
    time.sleep(0.05)

```

```

try:
    reply = httpx.get(f"http://127.0.0.1:{port}/summary", timeout=5)
    print("over real HTTP:", reply.status_code, "|", len(reply.json()), "sectors")
finally:
    server.should_exit = True          # ask the server to stop cleanly
    thread.join(timeout=10)
print("server has stopped:", not thread.is_alive())

# Files that usually travel with an application when it moves to another server
out = Path(__file__).parent / "_tmp" / "serving"
out.mkdir(parents=True, exist_ok=True)
(out
"requirements.txt").write_text("fastapi\nuvicorn\nstreamlit\npandas\nfolium\n",
encoding="utf-8")
commands = {
    "API (FastAPI)": "uvicorn api_unit:app --host 127.0.0.1 --port 8000",
    "Dashboard (Streamlit)": "streamlit run app_streamlit.py",
    "Demo (Gradio)": "python app_gradio.py",
    "Dashboard (Dash)": "python app_dash.py",
    "Light web (Flask)": "flask --app app_htmx run",
}
print("\nCommands that start each app (run them YOURSELF in a terminal, not here):")
for name, c in commands.items():
    print(f" {name:22} {c}")

```

```

over real HTTP: 200 | 5 sectors
server has stopped: True

```

```

Commands that start each app (run them YOURSELF in a terminal, not here):
API (FastAPI)          uvicorn api_unit:app --host 127.0.0.1 --port 8000
Dashboard (Streamlit)  streamlit run app_streamlit.py
Demo (Gradio)          python app_gradio.py
Dashboard (Dash)       python app_dash.py
Light web (Flask)      flask --app app_htmx run

```

The server is started on a free port, answers 200 with 5 sectors, then is stopped cleanly. The list below is the start command for each app in this chapter, which **you run yourself in a terminal**; nothing there is run here.

How do you move it so other people can use it? Here are the options named by **each official documentation**. We did not sign up, upload, or try any of them; terms, prices, and limits may change, so read the official page before deciding.

Application	Options according to the documentation
Streamlit	Streamlit Community Cloud (described as free hosting for non-commercial, personal, and educational apps), Snowflake , or other platforms through tutorials
Gradio	share=True (a temporary link from your computer), Hugging Face Spaces (a permanent link), or mounted on your own FastAPI app with <code>gr.mount_gradio_app()</code>
Dash	Plotly Cloud (managed) or Dash Enterprise (for organizations)
FastAPI, Flask	your own or rented server, running <code>uvicorn</code> as above

A note from the Hugging Face documentation worth reading carefully: at the time of writing, Spaces of type Gradio or Docker need a paid plan to create, while static Spaces are free, and a Space on free hardware “goes to sleep” when unused for a while. The documentation also advises keeping secrets (keys, tokens) as a settings **secret**, **not** writing them in code.

⚠ WARNING

Before serving anything to the public, ask yourself: (1) may everyone see this data? A dashboard without a password is open to anyone who has the address; (2) are there secrets in the code? Keys and passwords must **not** be in code or a repository; (3) who maintains it when it breaks? The chapters of Series A1 cover security and deployment seriously.

Running Project I5 (closing): one dataset, many doors

The Series I5 project is now complete: the business-unit data is read by one module, then opened through many doors.

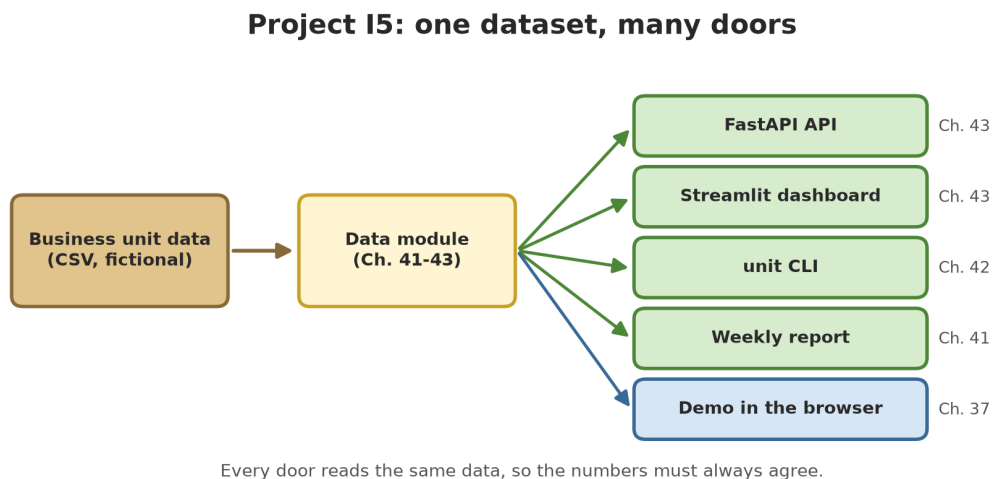


Figure 0.2: Project I5: one dataset, many doors.

- **FastAPI API** (this chapter) for other programs.
- **Streamlit dashboard per sector** (this chapter) for people.
- **The `unit` CLI** (Chapter 42) for the terminal and automation.
- **The weekly report** with polite scraping (Chapter 41).
- **The demo in the browser** without a server (Chapter 37), and maps (Chapters 38 to 40).

Because all the doors read the same data, the numbers **must agree**. The closing script checks that: the mean per sector is calculated directly from the module, through the API, and through the dashboard, then compared.

```
# Chapter 43 - Running project I5 (closing): one dataset, three ways in, the results
must agree
import logging

from fastapi.testclient import TestClient
from streamlit.testing.v1 import AppTest

from api_unit import app
from unit_data import read_units, summarize_by_sector

logging.getLogger("streamlit.runtime.scriptrunner_utils.script_run_context").setLevel(logging.ERROR)
```

```

direct = summarize_by_sector(read_units()) # way 1:
the data module
via_api = {r["sector"]: r for r in TestClient(app).get("/summary").json()} # way 2:
the API

dashboard = AppTest.from_file("app_streamlit.py", default_timeout=30).run()
print(f"{'sector':>10} {'units':>5} {'mean (module)':>14} {'API':>8} {'dashboard':>10}
match")
all_match = True
for sector, r in direct.items():
    dashboard.selectbox[0].select(sector).run() # way 3:
the Streamlit dashboard
    metrics = {m.label: m.value for m in dashboard.metric}
    match = (f"{r['mean']:.1f}" == f"{via_api[sector]['mean']:.1f}" == metrics["Mean
value"])
            and str(r["count"]) == metrics["Number of units"])
    all_match &= match
    print(f"{'sector':>10} {'count':>5} {'mean':>14.1f} {'via_api[sector]
['mean']:>8.1f} "
          f"{metrics['Mean value']:>10} {'yes' if match else 'NO'}")

print("\nall ways agree:", all_match)

```

sector	units	mean (module)	API	dashboard	match
forest	3	35.2	35.2	35.2	yes
plantation	4	257.5	257.5	257.5	yes
paddy	2	1115.2	1115.2	1115.2	yes
pond	3	369.3	369.3	369.3	yes
barn	3	383.8	383.8	383.8	yes

all ways agree: True

All five sectors match across the three ways (the last column says “yes”, and `all ways agree: True`). A check like this is cheap and a lifesaver: if one day a path starts calculating differently, you know **before your users do**.

Exercises

1. In `app_streamlit.py`, add an `st.slider` “Minimum value” that filters the table. Add a check in `02_streamlit_interaction.py` that moves the slider and reads the number of table rows.
2. In `api_unit.py`, add an access point `GET /sector` that returns the list of sector names, then test it with `TestClient`.
3. In `app_dash.py`, add a second component (for example `dcc.RadioItems` to choose ascending or descending order) and its callback. Test it through the Flask test client.
4. **Challenge:** make `app_gradio.py` accept a unit number (`D1`, `L2`, ...) and return that unit by calling your FastAPI API through `TestClient` (not over a network). Show a friendly error when the number does not exist.

Chapter Summary

- **Streamlit:** a script from top to bottom, re-run on every interaction; test with **AppTest** without a browser.
- **Gradio:** wraps a function/model with automatic boxes; test the function directly or through **gradio_client**. Do not use `share=True` for data that is not for the public.
- **Dash:** layout + callbacks, built on Flask; test with the test client.
- **folium** and **pydeck** produce interactive HTML maps that load their libraries from the internet when opened.
- **HTMX** + Flask: the server sends HTML fragments; only the server side was tested here.
- **FastAPI:** a JSON API with validation and automatic docs; test with `TestClient`.

- Serving = running on a machine users can reach; consider data, secrets, and maintenance first.

Closing Series I5

Congratulations! You have finished Series I5, and that means the **Intermediate level (Volume 2) is complete**. From tidy, maintained code (Series I1), data and the internet (I2), statistics and modelling (I3), advanced Python language features (I4), to first applications (I5), you can now:

- write code that is tidy, tested, and under version control;
- fetch, clean, and analyse data, including basic statistics;
- use generators, decorators, type hints, Pydantic, and async;
- build web APIs, process spatial data, and show it on maps;
- automate repetitive tasks, build command-line tools, and present data through dashboards.

But your application still **runs on your own computer**. For it to be **fast, safe, servable** to many users, able to **handle big data**, and **smart**, you need to step into the **Advanced** level: professional software engineering, large-scale data, machine learning, up to AI.

Next, in Chapter 44: performance. We learn to measure and speed up code (profiling, the right data structures, vectorization, and caching).

Level up: from small programs to **real systems**

- Tidy code: OOP, testing, Git, and code quality
 - Data and the internet: APIs, regex, databases, NumPy, pandas
 - Statistics, regression, and data analysis
 - Advanced Python, the web, maps (GIS), and dashboards
-

Volume 2 is for readers who already write small programs and want to work with data, maps, and web apps. Examples stay with forests, orchards, rice fields, ponds, and barns, using made-up data that is safe to run.

Compiled by

Badar Mubarak Yogaswara