PYTHON FROM ZERO SERIES

Python from Zero to Very Advanced

Volume 1 · Basic

Chapters 1–15

Badar Mubarok Yogaswara

Contents

Python from Zero to Very Advanced 9

About This Book 11
Who is this book for? 11
What is covered, and what is not 11
What we assume about you 11
How to read what is written in this book 11
Icons in this book 12
The series map: 13 series in three levels 12
Splitting into three volumes 13
How to read: in order or by jumping 14
Beyond the book 14
The Python version used 14
An honest note: what was tested and what was not 14
Ready? 15

Chapter 1 — Computers and Programs: The Super-Obedient Employee With No Common Sense 17
A computer is a very obedient employee 17
Three main jobs: input, process, output 17
Meet the computer's insides through a warehouse 18
What is a program? 19
Files and folders: the computer's drawer system 19
Why Python? 20
Exercises 20
Chapter Summary 20

Chapter 2 — Thinking Like a Programmer: Writing Recipes for a Computer 21
Algorithm: a recipe that must not be ambiguous 21
The three basic ingredients of every program 21
Flowcharts: drawing an algorithm 22
Break big problems into small steps 24
Testing an algorithm in your head 24
Exercises 24
Chapter Summary 24

Chapter 3 — Meet the Terminal: Chatting With Your Computer by Typing . . . 25
The terminal is like sending messages . . . 25
Opening the terminal . . . 25
Reading the "prompt" . . . 25
Six basic commands . . . 26
Practice: make a learning folder . . . 26
When you mistype: error messages . . . 27
Exercises . . . 27
Chapter Summary . . . 27

Chapter 4 — Meet Python and Install It: Your First Program . . . 29
What is Python? . . . 29
Installing Python 3.14 on Windows . . . 29
Checking the installation . . . 30
Trying Python directly: interactive mode . . . 30
Your first program . . . 31
Running Project B1: "Hello, Farm!" . . . 31
When there's an error: stay calm, it's normal . . . 32
Trap 1: mistyping a command name . . . 32
Trap 2: a missing closing quotation mark . . . 32
Exercises . . . 33
Chapter Summary . . . 33

Chapter 5 — Variables and Data Types: Giving Everything a Name . . . 35
A variable is like a label on a sack . . . 35
Naming rules (so Python doesn't sulk) . . . 36
Four data types you'll meet every day . . . 37
Printing neat sentences with f-strings . . . 37
A classic trap: numbers that are really text . . . 38
A typo? Python lends a hand . . . 40
Running Project B2: your first field data . . . 40
Exercises . . . 41
Chapter Summary . . . 41

Chapter 6 — Operators and Input/Output: Calculating, Comparing, and Chatting With Your Program . . . 43
Calculating like a calculator . . . 43
Three ways to divide . . . 44
Comparing values . . . 45
Combining conditions: and, or, not . . . 46
The order operators are carried out . . . 46
The decimal surprise . . . 47

Talking with the program: print() and input() . . . 48
Running Project B2: the pond calculator . . . 48
Exercises . . . 50
Chapter Summary . . . 50

Chapter 7 — Conditionals: Teaching a Program to Make Decisions . . . 51
if: do something only when the condition is true . . . 51
else: the second road . . . 51
elif: many roads . . . 52
Combining conditions . . . 53
Protecting a program from errors . . . 53
match and case: choosing one of many values . . . 54
A trap: forgetting to indent . . . 54
Running Project B2: pond status . . . 54
Exercises . . . 55
Chapter Summary . . . 55

Chapter 8 — Loops: Telling a Program to Do the Same Thing Many Times . . . 57
for: for every item in a list . . . 57
range: a run of numbers . . . 57
The accumulator pattern: adding up with a loop . . . 58
while: repeat while the condition is true . . . 59
break and continue . . . 59
A loop inside a loop . . . 60
A trap: range doesn't accept decimals . . . 60
Running Project B2: plantation harvest summary . . . 60
Exercises . . . 61
Chapter Summary . . . 61

Chapter 9 — Built-in Data Structures: Containers for Tree, Fish, and Livestock Data . . . 63
list: a list you can change . . . 63
Slicing: taking a piece . . . 64
tuple: a locked list . . . 65
dict: labelled data . . . 65
set: a collection without duplicates . . . 66
Comprehension: a new list in one line . . . 66
Running Project B2: tree data . . . 67
Exercises . . . 67
Chapter Summary . . . 68

Chapter 10 — Functions: Write Once, Use Many Times 69
def: making a function 69
Arguments and return 70
Default values 70
Scope: who can see which variable 71
Returning several values 71
Docstring and lambda 72
A classic trap: forgetting return 72
Running Project B2: the field data recorder 72
Exercises 73
Chapter Summary 73
Closing Series B2 73

Chapter 11 — Modules and Packages: Borrowing Tools from Python's Warehouse 75
Three ways to use import 75
math and random 76
Dates with datetime 77
statistics and collections 77
itertools and pathlib 78
Making your own module 79
`if __name__ == "__main__"` 79
Packages: a folder of modules 80
pip: borrowing from outside the warehouse 81
Running Project B3: summary of five sectors 81
Exercises 82
Chapter Summary 82

Chapter 12 — Error Handling: Programs That Don't Collapse Easily 83
try and except 83
Know your error types 84
else and finally 85
raise: raising your own error 85
Your own error types 86
Reading a traceback 86
Hunting bugs: print, assert, logging 87
Running Project B3: resistant to wrong input 89
Exercises 91
Chapter Summary 91

Chapter 13 — Files and Basic I/O: Data That Survives Closing the Program 93
Writing a text file 93
Reading a text file 94
Modes: w, a, x 95
Why with 95
pathlib for files and folders 96
Encoding: when text turns into strange symbols 97
A file that does not exist 98
Running Project B3: saving and loading tree data 99
Exercises 100
Chapter Summary 100

Chapter 14 — Data Formats: CSV, JSON, and Be Careful with Pickle 101
CSV: a table as text 101
Reading CSV: everything is text 102
Why not `split(",")` 103
JSON: nested data 103
JSON to a file 104
Two JSON problems 105
Pickle: handy, but dangerous 105
Running Project B3: save and load five sectors 107
Exercises 108
Chapter Summary 108

Chapter 15 — Office Document Automation: From Data to Excel and PDF Reports 111
Installing the libraries 111
Excel with openpyxl 112
Charts inside Excel 114
Excel with pandas 114
Word with python-docx 115
PDF: making with reportlab 116
PDF: reading and pulling out tables 117
PDF: merging and splitting 117
Email: building a message 118
Email: testing against a local server 119
Running Project B3: field summary becomes a report 120
Exercises 123
Chapter Summary 124
Closing Series B3 124
Congratulations: Basic Level (Volume 1) is complete! 124

Appendix A — Installing on Windows, Linux, and macOS 125
Honesty first: what was tested and what was not . . . 125
The terminal: where you type commands . . . 125
Decision map: which way to install? . . . 126
Windows . . . 126
The Python install manager . . . 126
Results on the author's computer . . . 127
Checking the version . . . 127
The PATH folders . . . 127
Installing from a script (optional) . . . 127
Linux . . . 128
Python from distribution packages . . . 128
pyenv: installing many Python versions . . . 128
Building from source . . . 128
macOS . . . 129
The installer from python.org . . . 129
Homebrew . . . 129
uv: the shortcut on every system . . . 129
Virtual environments (venv) on each system . . . 130
A real test: creating a venv from a script . . . 130
Checking your environment . . . 131
Other tools this book uses . . . 132
Common problems and their solutions . . . 132

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 Mubarok 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 Mubarok 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 `kode/` 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

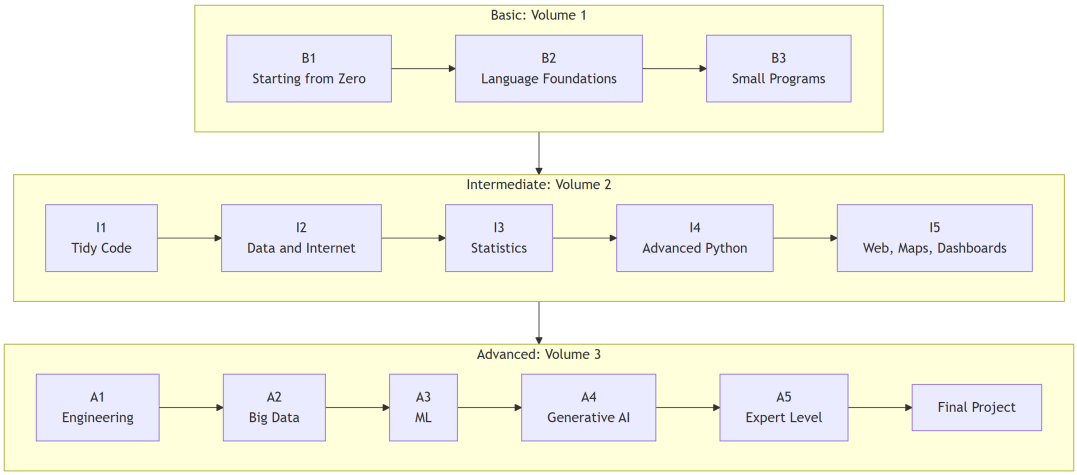
The book has **68 chapters**. The chapters are grouped into **13 series**, and the series into **three levels**. Each series has prerequisites, one running project, and a "bridge" to the next series.

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 Modelling	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.

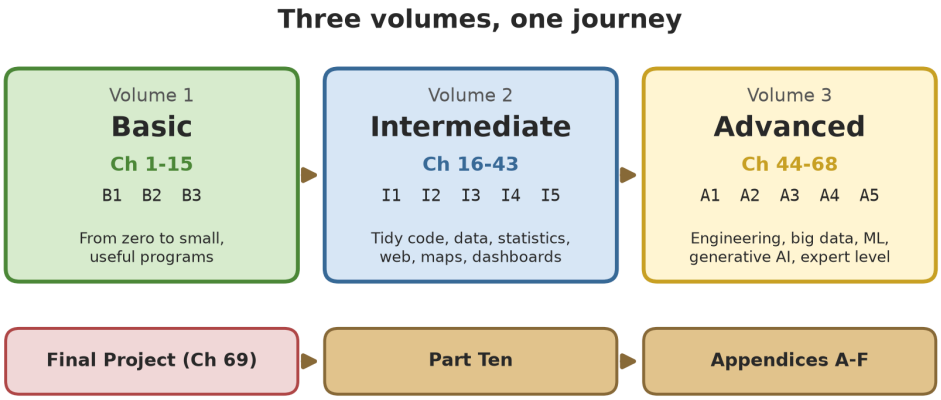


Figure 0.1: Three volumes, one journey.

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 1 — Computers and Programs: The Super-Obedient Employee With No Common Sense

Series B1 — Starting From Zero

In this chapter, you will:

- Understand what a computer really does (spoiler: only three things)
- Meet the parts of a computer through the analogy of a warehouse and a workbench
- Learn what a program, a file, a folder, and a file path are
- Be ready for Chapter 2 without any fear

This chapter has not a single line of code. Don't worry: this is exactly where the most important foundation is laid. People who understand *what a computer does* find it much easier to learn to program one.

A computer is a very obedient employee

Imagine you hire a warehouse worker with three odd traits:

1. **Extremely fast.** They can carry out millions of steps in one second.
2. **Extremely careful.** They never get tired, never get bored, and never miscalculate.
3. **No common sense.** They do *exactly* what you write down. No less, no more.

That is a computer. If you write "move that sack" without saying which sack, it won't guess. It stops and gets confused (or does something strange).

ℹ REMEMBER

A computer is **not smart**. It is only **very fast and very obedient**. All the "intelligence" you see comes from instructions written by people.

The good news: because a computer is obedient, when the result is wrong, it is almost always the instructions that are wrong, not the computer. And instructions can be fixed.

Three main jobs: input, process, output

Whatever a computer does, from playing a song to calculating salaries, it can always be broken into three stages:

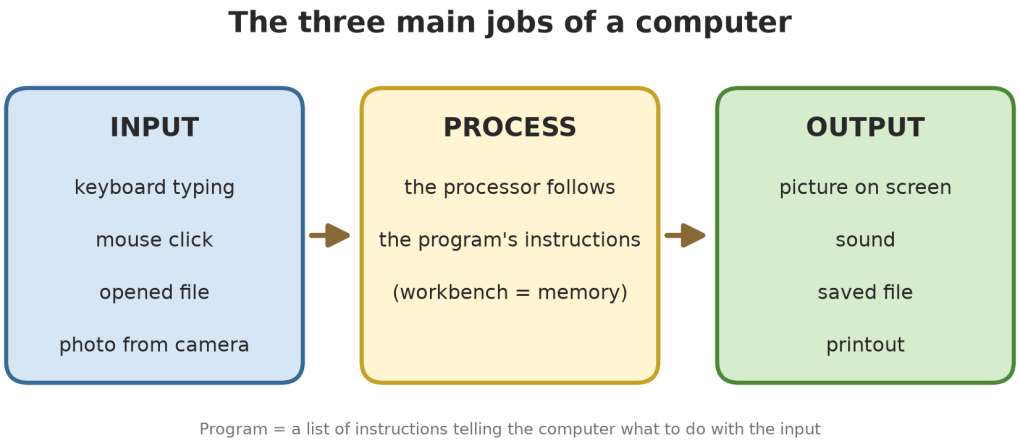


Figure 0.1: The three main jobs of a computer: take in input, process it by following the program's instructions, then produce output.

Stage	Meaning	Example from the field
Input	Data going into the computer	You type a tree diameter, `32.5`
Process	The computer works on that data	It calculates the trunk's cross-section area
Output	The result the computer produces	The number appears on screen, or is saved to a file

So when you learn Python later, you are really learning to write the **middle part** of this diagram: which instructions the computer should carry out on the input.

Meet the computer's insides through a warehouse

You don't need to be a technician. Just know these four parts:

Part	Analogy	What it does
Processor (CPU)	The warehouse worker	Carries out instructions one by one, very fast
Memory (RAM)	The workbench	A temporary place for things being worked on *right now*. Quick to reach, but **cleared when the computer is switched off**
Storage (SSD/hard disk)	The filing cabinet	Keeps files permanently. Slower to reach, but **contents last** even without power
Screen, keyboard, mouse	Windows and doors	The way in (input) and the way out (output)

> TIP
>
> Ever lost your typing because you forgot to press *Save* and the power went out? That's because your work was still on the "workbench" (RAM) and had not been moved to the "filing cabinet" (storage). **Save often.**

What is a program?

A **program** is a list of instructions written neatly so a computer can do something. Think of a recipe: ingredients go in, steps are followed in order, a dish comes out.

Almost everything you use every day is a program: your web browser, the calculator, your messaging app, even Windows itself. Without programs, a computer is just a warm, silent box.

A person who writes programs is called a **programmer**. And after reading this book, that will be you.

Files and folders: the computer's drawer system

Because storage holds thousands of items, the computer organises them with two simple ideas:

- **File** = one named collection of data. It could be a photo, a document, a song, or (soon) your own Python program.
- **Folder (directory)** = a drawer that holds files and other folders. Since a folder may contain more folders, the result looks like nested drawers.

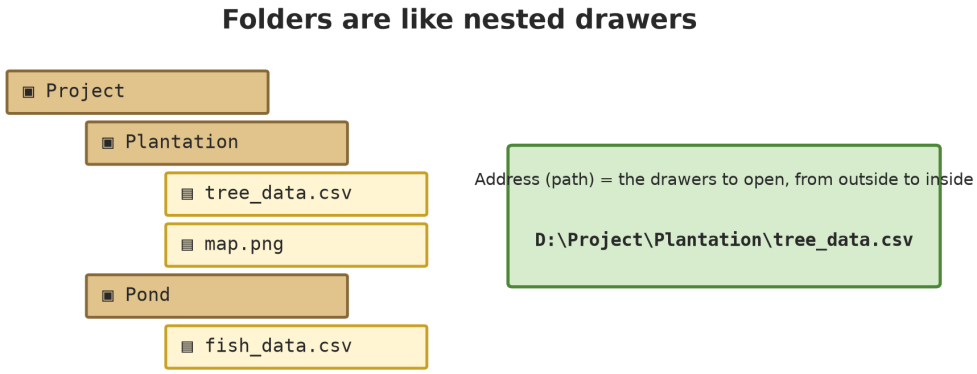


Figure 0.2: Nested folders are like drawers. A file's address (path) is the list of drawers to open, from outside to inside.

A file's **address** (called a *path*) shows the way to a file, from the outermost "drawer" down to the file name. On Windows, for example:

```
D:\Project\Plantation\tree_data.csv
```

Read it as: *on drive D, open the folder* `Project`*, then the folder* `Plantation`*, then take the file* `tree_data.csv`. The `\` sign separates one drawer from the next.

The part after the last dot in a file name, such as `.csv`, `.jpg`, or `.py`, is the **extension**. It tells the computer what kind of file it is and which program should open it. Your Python files will end in `.py`.

> ⚠ WARNING
>
> Windows often **hides extensions** by default, so `notes.txt` looks like just `notes`. This traps beginners: you think you created `hello.py`, but its real name is `hello.py.txt`. In File Explorer, turn extensions on via **View** > **Show** > **File name extensions**. Exercise 3 below walks you through it.

> 🔥 TECHNICAL STUFF (feel free to skip)
>
> At the deepest level, a computer knows only two states: current **on** or **off**, written as `1` and `0`. Letters, photos, and songs are all eventually stored as very long strings of 1s and 0s. You **do not** need to memorise this; Python hides all of that detail from you.

Why Python?

Computers only understand numbers. People only understand human language. A translator is needed: a **programming language**. Python is one of the programming languages whose writing looks most like human language, so it suits beginners, yet it is powerful enough for data scientists. We start that story in Chapter 4.

Exercises

1. Name **three inputs** and **three outputs** on your phone. (Hint: a touchscreen is input *and* output.)
2. Open File Explorer and write down the full address (*path*) of your `Documents` folder. (Hint: click the address bar at the top of the window.)
3. **Challenge:** turn on file name extensions, then find one `.jpg` or `.png` file on your computer. What is the extension of your Word documents?

Chapter Summary

- A computer = an employee who is **very fast, very careful, but has no common sense**: it follows instructions exactly.
- All computer work breaks down into **input → process → output**.
- **RAM** = workbench (temporary), **storage** = filing cabinet (permanent). Save your work often.
- **Program** = a list of instructions. **File** = named data, **folder** = drawer, **path** = address to a file.
- Turn on **extension** display so file names don't fool you.

Next, in Chapter 2: before writing instructions for this obedient employee, we practise writing clear instructions, starting with everyday jobs like feeding fish.

Chapter 2 — Thinking Like a Programmer: Writing Recipes for a Computer

Series B1 — Starting From Zero

In this chapter, you will:

- Understand what an algorithm is (spoiler: a very careful recipe)
- Meet the three basic ingredients of every program: sequence, choice, repetition
- Draw and read a simple flowchart
- Break a big problem into small steps

Still no code in this chapter. Good programmers **think first and type later**. This chapter trains that thinking muscle using field work you already know.

Algorithm: a recipe that must not be ambiguous

An **algorithm** is a clear sequence of steps to reach a goal. You already use them without noticing: cooking recipes, directions, how to fill in a form.

The difference from ordinary instructions: an algorithm for a computer **must not be ambiguous**. Remember from Chapter 1 that a computer is an employee with no common sense. Compare:

Instruction for a person	Instruction for a computer
"Feed the fish enough."	"Scatter 2 kg of feed. While the fish are still scrambling, add 200 g."
"Harvest when they're big."	"Harvest when the average weight is 500 g or more."
"Water the plants early in the morning."	"Water at 06:00 for 10 minutes."

⚠ WARNING

Words like *enough*, *about*, *rather*, and *when it's big* are easy for people but **impossible for a computer to carry out**. Whenever you write an instruction, ask: *"Could someone who knows nothing at all do this correctly?"*

The three basic ingredients of every program

A secret rarely told: a program of any size, from a calculator to a map app, is built from just **three basic ingredients**.

Ingredient	Meaning	Example from the field
Sequence	Steps done one by one, top to bottom	Take feed → scatter feed → record the amount
Choice	*If* a condition holds, do A; *otherwise*, do B	If the water pH is below 6, add lime; otherwise, leave it
Repetition	Repeat a step *while* a condition still holds	While the fish are still scrambling, add a little more feed

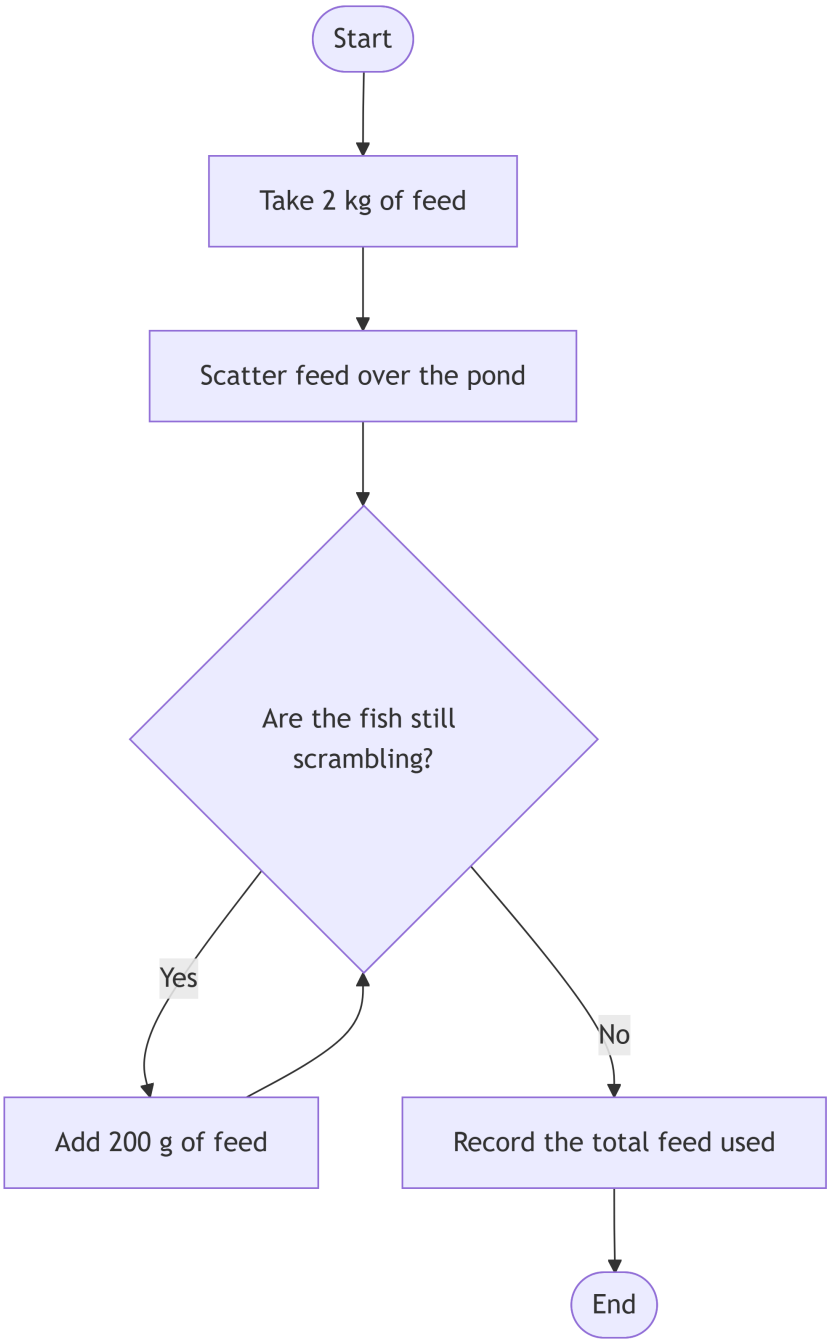
That's all. The rest of this book is essentially how to write these three things in Python, then apply them to data from forests, plantations, rice fields, ponds, and barns.

Flowcharts: drawing an algorithm

A **flowchart** is a way of drawing an algorithm so it is easy to see. There are only a few shapes:

Shape	Meaning
Oval	**Start** or **End**
Rectangle	One **step** to carry out
Diamond	One **question** answered "Yes" or "No"
Arrow	Direction of flow, to the next step

Here is the algorithm for feeding a pond, using all three ingredients at once:



The fish-feeding algorithm: sequence, choice, and repetition in one picture.
Trace it with your finger: start at the **Start** oval and follow the arrows. At the diamond you must choose "Yes" or "No". The "Yes" road **goes back** to the same question. That is **repetition**: looping until the answer is "No".

> TIP
>
> No special software is needed. A flowchart can be drawn on paper with a pencil. If you can trace your drawing without getting lost, the algorithm is clear enough to turn into code.

Break big problems into small steps

Suppose your boss asks: **"Work out this month's plantation harvest."** It sounds big and vague. Programmers break it down until each piece is clear:

1. Gather the harvest records of every plantation block.
2. For **each** block: add up the harvest weight of all days.
3. Add up the results of all blocks.
4. Compare with this month's target.
5. Display the result neatly.

Each number can be broken down further, until the smallest step is so clear it can be written as one or two lines of code. This habit of **breaking problems down** is worth more than memorising any language rule.

Testing an algorithm in your head

Before writing code, test the algorithm with small numbers. Example: 2 kg of feed, and the fish scramble twice before calming down.

Step	Feed scattered so far	Fish still scrambling?
First scatter	2000 g	Yes
Add 200 g	2200 g	Yes
Add 200 g	2400 g	No

Result: a total of **2400 g**. Testing by hand like this is called a **dry run**. If the result already makes sense on paper, only then are we confident to type it.

TECHNICAL STUFF (feel free to skip)

Besides flowcharts, programmers often write algorithms as **pseudocode**: everyday sentences arranged like code, for example "REPEAT while fish scramble: add 200 g". Pseudocode is not run by a computer. It is only a thinking tool before writing real code.

Exercises

1. Write an algorithm for **making sweet tea** in six or more steps. Make sure it contains no word like *enough*.
2. Draw a flowchart for: *"Check the pond water pH. If it is below 6, add lime and check again. If it is 6 or more, record the result."* Which part is a repetition?
3. **Challenge:** break the task *"Count the healthy trees in one forest plot"* into at least five steps, then mark which steps are choices and which are repetitions.

Chapter Summary

- **Algorithm** = a clear, unambiguous sequence of steps to reach a goal.
- All programs are built from three ingredients: **sequence**, **choice**, **repetition**.
- A **flowchart** draws an algorithm: oval (start/end), rectangle (step), diamond (question), arrow (direction).
- **Break** big problems into small steps, then do a **dry run** on paper before typing.

Next, in Chapter 3: we meet the *terminal*, the black window where you "chat" with the computer by typing, with nothing to fear.

Chapter 3 — Meet the Terminal: Chatting With Your Computer by Typing

Series B1 — Starting From Zero

In this chapter, you will:

- Know what the terminal is and why programmers use it
- Open a terminal and read its "prompt"
- Move between folders, list a folder's contents, and make a new folder
- Read terminal error messages without panicking

Many beginners find the terminal scary: a black window, white text, and supposedly one typo ruins everything. The reality is much tamer. By the end of this chapter you will have made a working folder for learning Python, using the terminal, and you will smile at how "scary" it once seemed.

The terminal is like sending messages

So far you have used the computer by **clicking**: click a folder, click a file, click a button. The terminal is another way: you **type a command**, and the computer **answers in text**. It is like messaging the warehouse worker from Chapter 1: you write a request, they reply.

Why do programmers like the terminal?

- **Fast.** One typed line can replace a dozen clicks.
- **Repeatable.** The same command can be used again without repeating the same clicks.
- **Used everywhere.** Servers, data tools, and nearly all programming tools can be controlled from a terminal.
- **Python runs from here.** From Chapter 4 on, you will run programs through the terminal.

Opening the terminal

The examples in this book use **Windows** and a program called **PowerShell**.

- **Windows:** press the *Start* button, type `Terminal` (or `PowerShell`), then press Enter.
- **macOS:** open the *Terminal* app via Spotlight (Cmd + Space).
- **Linux:** the *Terminal* app is usually in the applications menu; the exact way differs between distributions.

The basic commands in this chapter work similarly on all three, but some names and slash directions differ. The book's examples follow Windows PowerShell.

Reading the "prompt"

When the terminal opens, you see one blinking line waiting for you to type, roughly like this (what you see on your computer will differ):

```
PS C:\Users\You>
```

This line is the **prompt**, meaning "the computer is ready for a command". The important part is **`C:\Users\You`**: that is **the folder you are currently standing in**. Picture yourself standing in front of one drawer; the commands you type work on that drawer.

> **REMEMBER**
>
> The terminal always "stands" in one folder. If a command doesn't work as expected, the first question to ask is: **"Which folder am I standing in?"**

Remember the nested drawers from Chapter 1? Every command below works on one of those drawers:

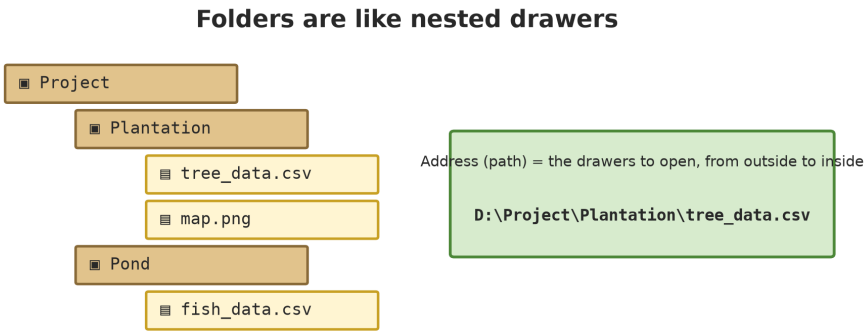


Figure 0.1: Nested folders: the terminal always "stands" in one drawer, and the `cd` command moves between drawers.

Six basic commands

Type the command then press **Enter**. (Anything inside `< >` is a name you replace yourself; don't type the < and > signs.)

Command	Meaning	Analogy
`pwd`	Show the folder I'm standing in	"Which drawer am I at?"
`dir`	Show what's inside this folder	Peeking into the drawer
`cd <folder-name>`	Go into that folder	Opening a deeper drawer
`cd ..`	Go up one level (two dots)	Stepping back out to the outer drawer
`mkdir <folder-name>`	Make a new folder	Adding a new drawer
`cls`	Clear the screen	Wiping the whiteboard (folder contents are not erased)

> **TIP**
>
> Two shortcuts you will often be thankful for. The **up arrow** repeats the previous command. The **Tab** key completes folder names: type `cd Docu` then press Tab, and the full name appears by itself.

Practice: make a learning folder

Let's make a working folder for this whole book. Type these one by one (press Enter after each line):

```
cd Documents
mkdir learn-python
cd learn-python
pwd
```

After the last line, the terminal shows the address of your new folder, ending in `learn-python`. Congratulations: you just made a folder and went inside it without a single mouse click.

To check, type `dir`. The folder is still empty, so nothing is listed. This is the folder we will fill with Python programs in Chapter 4.

> ⚠ WARNING
>
> If your *Documents* folder is synced with **OneDrive**, the folder you see in File Explorer may be somewhere other than `C:\Users\You\Documents`, for example `C:\Users\You\OneDrive\Documents`. On the author's computer, both exist. That is fine for learning, **as long as you always use the same folder**. Write down your `pwd` result, then open that folder in File Explorer so you know where it really is.

When you mistype: error messages

Try getting it wrong on purpose. Type a folder name that doesn't exist:

```
cd Docs
```

The terminal answers with a message like this (the folder name on your line will differ; in PowerShell 7.6.6 this message is preceded by the command name):

```
Cannot find path 'C:\Users\You\Docs' because it does not exist.
```

It means: *"Can't find that address because it really doesn't exist."* Nothing is broken. The message even gives the exact name that couldn't be found. The cure: type `dir` to see the correct folder names (for example `Documents`), then try again.

> ⚠ WARNING
>
> The terminal has no "Undo" button for **delete commands** (for example `del`, `rm`, or `Remove-Item`). This book does not teach them yet. A safe rule: **never copy commands from the internet that you don't understand**, and if in doubt, ask first. As long as you use the six commands above, nothing can be damaged.

> 💡 TIP
>
> Command stuck, or the terminal seems "frozen"? Press **Ctrl + C** to cancel the running command.

> 🔥 TECHNICAL STUFF (feel free to skip)
>
> The word "terminal" really refers to the *window*, while the program that interprets what you type is called a **shell**. PowerShell, Command Prompt (`cmd`), and `bash` are examples of shells. The commands in this chapter were chosen on purpose because they are the same in many shells, so they are easy to carry over.

Exercises

1. Open the terminal and type `pwd`. Write down the address that appears, then match it to the folder name you see in File Explorer.
2. Inside the `learn-python` folder, make three folders: `ch04`, `ch05`, and `project`. Use `dir` to prove it. Then go back to the parent folder with `cd ..`.
3. **Challenge:** make a folder `plantation`, go into it, make a folder `block-a`, go into `block-a`, then return to `learn-python` using only the `cd ..` command. How many times did you type `cd ..`?

Chapter Summary

- **Terminal** = a window for typing commands; the computer answers in text.

- The **prompt** shows **the folder you are standing in**.
- Six basic commands: `pwd`, `dir`, `cd <folder>`, `cd ..`, `mkdir <folder>`, `cls`.
- **Up arrow** repeats a command, **Tab** completes names, **Ctrl + C** cancels.
- Error messages say what is wrong; read them slowly, don't panic.

Next, in Chapter 4: we meet Python, install it, then run your first program in the `learn-python` folder you just made.

Chapter 4 — Meet Python and Install It: Your First Program

Series B1 — Starting From Zero

In this chapter, you will:

- Know what Python is and what it is used for
- Install Python 3.14 and check that the installation worked
- Try Python directly through interactive mode
- Write and run your first program: "Hello, Farm!"
- Read Python error messages without panicking

All scripts for this chapter are in the `kode/ch04/` folder. On the web version, you can also download them all at once (zip).

What is Python?

Python is a programming language: a language people use to write instructions for a computer (remember the warehouse worker from Chapter 1). Its name, reportedly, comes from the comedy group *Monty Python*, not from the snake.

Why is Python a good fit for you?

- **Easy to read.** It looks like simple English, so beginners understand quickly.
- **Versatile.** Used for data analysis, automating work, mapping (GIS), websites, artificial intelligence, and much more.
- **Free and open.** Anyone may install and use it.
- **Big community.** Almost every beginner problem has already been asked and answered by someone.

Installing Python 3.14 on Windows

This book uses **Python 3.14**. The official Python documentation recommends installing on Windows through the **Python install manager**; the old single-file installer has been marked deprecated since version 3.14.

The steps:

1. Open python.org/downloads and download the **Python install manager** for Windows (or get it from the Microsoft Store).
2. Run the downloaded file and follow the on-screen instructions.
3. Open a terminal (Chapter 3), then type:

```
py install 3.14
```

This asks the manager to install Python 3.14. If questions appear, read and follow them. Adding a folder to `PATH` is **optional** according to the documentation; the `py` command works without it.

⚠ WARNING

If your computer once had Python installed the old way, there may be **more than one** program called `py.exe`: the old launcher and the new manager. On the author's computer there are three. The old launcher does not know the manager's commands: `py list` and `py install --help` are read as a *file name* and answered with a message like `can't open file ... list`. If that happens to you, use the unambiguous name **`pymanager`**: `pymanager install 3.14` and `pymanager list`. According to the official documentation, `pymanager` exists precisely to avoid clashing with an old `py.exe`. To list every `py.exe` on your computer: `where.exe py`.

ℹ REMEMBER

Installing on macOS and Linux works differently. The official instructions are at python.org/downloads. This book uses Windows as its example, but almost all example programs behave the same on other systems.

Checking the installation

The installation worked if this command shows a version number:

```
py --version
```

On the author's computer, the result is:

```
Python 3.14.8
```

The last number (for example `.8`) may differ on your computer; what matters is that it starts with `3.14`.

💡 TIP

This book tests all examples with **Python 3.14.8**. All scripts in this book were also tried on Python 3.12.10: the results are the same, except that in Python 3.12 error messages show the full file path (not just the file name as in this book) and the pointer marks under the error line sometimes look different.

🔥 TECHNICAL STUFF (feel free to skip)

Anaconda is an alternative installation package that brings Python together with many data science libraries. Many data scientists use it. You **do not need** Anaconda to read Series B1 and B2. Its explanation and comparison are in Chapter 17.

Trying Python directly: interactive mode

Python has a "chat room" for trying one line of code and seeing the answer at once. It is called **interactive mode** (also known as the REPL). In the terminal, type:

```
py
```

A `>>>` sign appears, meaning Python is ready for code. Type `2 + 3` then Enter:

```
>>> 2 + 3
5
>>> print("Hello")
Hello
```

Python calculates `2 + 3` and answers `5` right away. To leave, type `exit()` then Enter.

Interactive mode is good for quick experiments. But anything typed there is **not saved**. For a program you can use again, we write it in a file.

Your first program

The steps are simple:

1. **Write** the code in a text editor and save it as a `.py` file.
2. **Run** that file through the terminal.
3. **Read** the result (and fix it if there's an error).

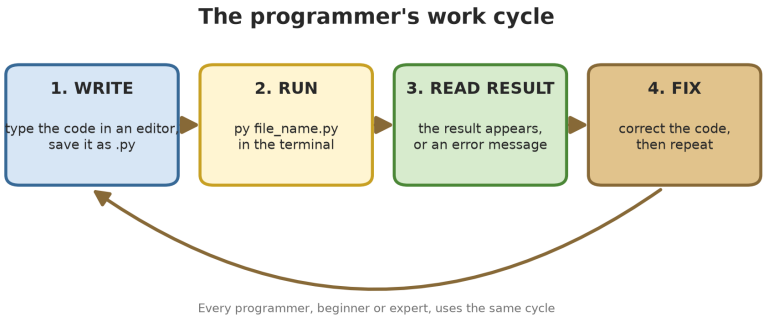


Figure 0.1: The programmer's work cycle: write, run, read the result, fix, repeat.

To type code, **Notepad** is enough. If you want more comfort (coloured code, automatic suggestions), many beginners use **Visual Studio Code**, a free editor.

> ⚠ WARNING
>
> When saving, make sure the file name ends in **`.py`** and not `.txt`. In Notepad, change "Save as type" to *All files*, then type the full name such as `hello.py`. If extensions in File Explorer are still hidden (Chapter 1), you may unknowingly create `hello.py.txt`.

Create a file named `hello.py` in the `learn-python` folder (from Chapter 3), with this content:

```
# Chapter 4 - First program: say hello to the world
print("Hello, World!")
```

Run it from the terminal (make sure you are standing in the `learn-python` folder):

```
py hello.py
```

The result:

```
Hello, World!
```

Congratulations, you just ran your first program. Let's take it apart:

- `print(...)` is a **command** meaning "show this on the screen".
- The text you want to show is wrapped in **quotation marks** `"..."`.
- Text inside the quotes appears exactly as written, without the quotes themselves.

Running Project B1: "Hello, Farm!"

Throughout this book we build one project: an **Information System for Forestry, Plantations, Farming, Fisheries, and Livestock**. Its first step is simple: a program that greets and shows the name and location of a business unit.

```
# Chapter 4 - Running project B1: "Hello, Farm!"
# A line that starts with # is a comment: Python ignores it.
```

```
print("Hello, Farm!")
print("Business unit: Sejahtera Teak Plantation")
print("Location: Bogor Regency, West Java")
```

```
Hello, Farm!
Business unit: Sejahtera Teak Plantation
Location: Bogor Regency, West Java
```

Notice the lines starting with `#`. That is a **comment**: a note for humans that Python ignores. Comments help you (and your colleagues) remember the purpose of code three months from now.

> REMEMBER
>
> Python reads a program **from top to bottom**, one line at a time. The order of the lines decides the order of results on the screen.

When there's an error: stay calm, it's normal

Even experienced programmers see errors every day. An error is not a sign you failed; it is Python's way of telling you **what needs fixing**. The two most common beginner traps:

Trap 1: mistyping a command name

```
# Chapter 4 - A typo: print written as "prnt"
prnt("Hello, Farm!")
```

```
Traceback (most recent call last):
  File "03_typo.py", line 2, in <module>
    prnt("Hello, Farm!")
    ^^^^
NameError: name 'prnt' is not defined. Did you mean: 'print'?
```

Read from the **last line**: `NameError` means Python doesn't recognise the name `prnt`. And Python itself guesses what you meant: *"Did you mean: 'print'?"* Fix the spelling, done.

Trap 2: a missing closing quotation mark

```
# Chapter 4 - The closing quotation mark is missing
print("Hello, Farm!)
```

```
  File "04_missing_quote.py", line 2
    print("Hello, Farm!)
          ^
SyntaxError: unterminated string literal (detected at line 2)
```

`SyntaxError` means a **writing rule** was broken. This time Python points at text whose opening quote has no partner. Add the closing `"`, done.

> TIP
>
> Read error messages **from the bottom up**: the last line explains the *kind* of problem, the lines above point to *where*. Three months from now, this habit will save you hours of frustration.

⚠ WARNING

Don't name your file the same as a built-in Python command or library, for example `print.py` or `random.py`. Python may get confused about which one you mean. Use a clear name such as `hello_farm.py`.

Exercises

1. Install Python 3.14, then run `py --version`. Write down the version number that appears.
2. Write your own `hello.py`, but change the content to your name and your city. Run it through the terminal.
3. Change `hello_farm.py` so it shows **three** different business units (for example a plantation, a pond, and a barn), each with its location.
4. **Challenge:** deliberately delete one bracket in one of the `print` lines. Read the error message that appears from the last line. What does it say?

Chapter Summary

- **Python** = an easy-to-read, versatile, free programming language.
- This book uses **Python 3.14**; check it with `py --version`.
- **Interactive mode** (`py`) for trying a single line; **`.py` files** for programs that are saved.
- Work cycle: **write → run → read the result → fix**.
- `print("...")` shows text; lines starting with `#` are comments.
- Python reads a program **from top to bottom**.
- Errors are hints, not punishments: read **from the last line**.

Next, in Chapter 5 (Series B2): your program can run, but it cannot *remember* anything yet. We learn variables and data types, so a program can store a tree's name, a cow's weight, and the number of fish.

Chapter 5 — Variables and Data Types: Giving Everything a Name

Series B2 — Python Language Foundations

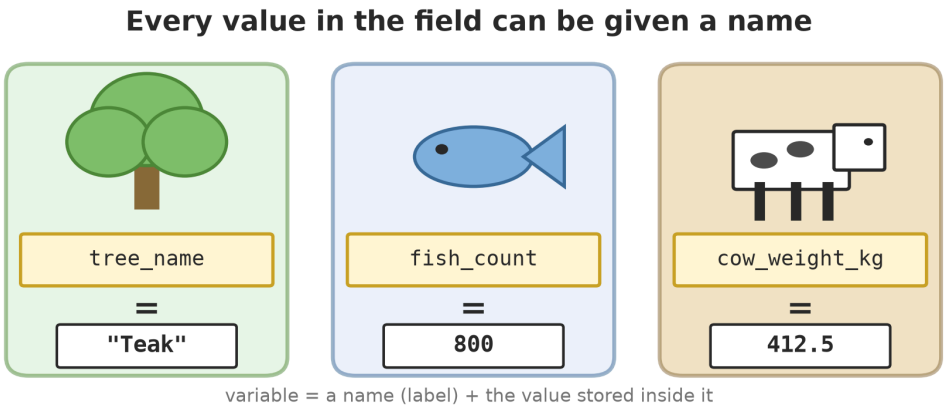


Figure 0.1: Opening picture: every value in the field (tree, fish, cow) is given a name, like a label on a sack.

In this chapter, you will:

- Understand what a variable is (spoiler: it's like a label on a feed sack)
- Meet the four data types you'll use most often
- Print neat sentences with f-strings
- Read error messages without panicking

All scripts for this chapter are in the `kode/ch05/` folder. On the web version, you can also download them all at once (zip).

A variable is like a label on a sack

Picture the feed store on a farm. There are dozens of sacks that look exactly alike. Without labels, you'd have to open every sack to find out what's inside. What a hassle!

So you stick on labels: **"Cattle Concentrate"**, **"Fish Pellets"**, **"Rice Bran"**. Now you just read the label.

A **variable** in Python works exactly like that. A variable is a *name* you attach to a *value*, so you can easily call that value up again later.

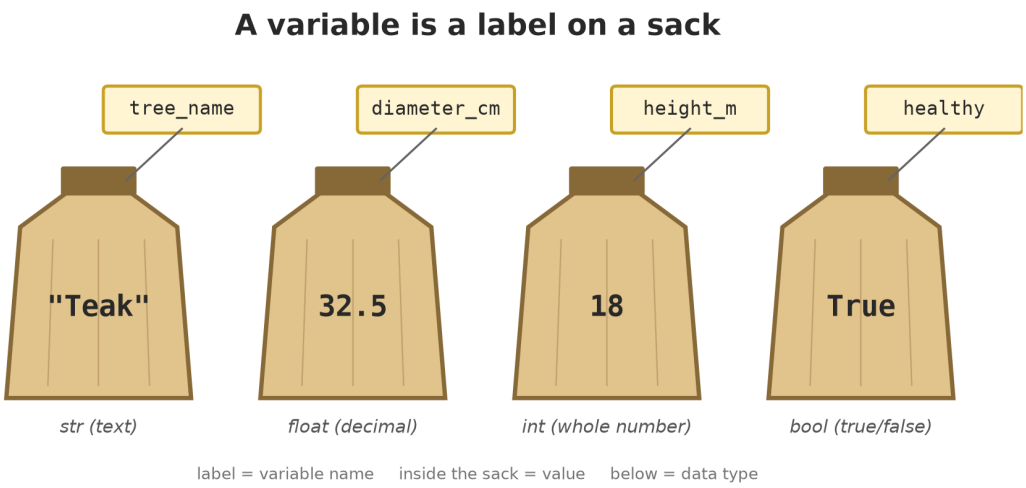


Figure 0.2: A variable is like a label on a sack: the variable name is on the label, the value is inside the sack, and every value has a data type.

```
# Chapter 5 - Creating variables: putting a "label" on a value
tree_name = "Teak"
diameter_cm = 32.5
height_m = 18
healthy = True
print(tree_name, diameter_cm, height_m, healthy)
```

The result:

```
Teak 32.5 18 True
```

The `=` sign here does **not** mean "equals" as in math class. Read it as **"store into"**: *store "Teak" into `tree_name`*.

> **i** REMEMBER
>
> In Python, `=` means *store*. To *compare* whether two values are equal, you use `==` (two equals signs). That's covered in Chapter 6.

Naming rules (so Python doesn't sulk)

Python is a bit picky about variable names. Here are the rules:

1. Letters, digits, and the underscore `_` are allowed.
2. A name **can't** start with a digit. `1tree` is rejected, `tree1` is fine.
3. **No** spaces. Use underscores: `cow_weight`, not `cow weight`.
4. Upper and lower case are different. `Weight` and `weight` are two different variables.

> TIP
>
> Choose names that describe the contents, **including the unit**: `diameter_cm`, `height_m`, `avg_weight_g`. Three months from now, you (or your colleague) will be grateful not to have to guess whether `d = 32.5` means centimetres, inches, or the tree's age.

Four data types you'll meet every day

Every value in Python has a **type**, meaning the kind of data it is. Python can tell you the type with the `type()` function:

```
# Chapter 5 - Checking data types with type()
tree_name = "Teak"
diameter_cm = 32.5
height_m = 18
healthy = True
print(type(tree_name))
print(type(diameter_cm))
print(type(height_m))
print(type(healthy))
```

```
<class 'str'>
<class 'float'>
<class 'int'>
<class 'bool'>
```

Type	Full name	What it holds	Field example
`str`	*string* (text)	Letters, words, sentences; always in quotes	`"Teak"`, `"Pond A"`, `"Bali cattle"`
`int`	*integer* (whole number)	Numbers without a decimal point	`18` trees, `1250` fish
`float`	decimal number	Numbers with a decimal point	`32.5` cm, `85.4` grams
`bool`	*boolean* (true/false)	Only `True` or `False`	`healthy = True`, `harvested = False`

> ⚠ WARNING
>
> Python uses a **dot** for decimals, never a comma: write `32.5`, not `32,5`. If your country writes decimals with a comma, this is one of the most common beginner traps.

Printing neat sentences with f-strings

The easiest way to mix text and numbers is an **f-string**: put the letter `f` before the opening quote, then place variables inside curly braces `{ }`.

```
# Chapter 5 - Printing neat sentences with f-strings
fish_count = 1250
avg_weight_g = 85.4
print(f"Pond A holds {fish_count} fish, average weight {avg_weight_g} grams.")
total_kg = fish_count * avg_weight_g / 1000
print(f"Estimated harvest: {total_kg:.1f} kg")
```

```
Pond A holds 1250 fish, average weight 85.4 grams.
Estimated harvest: 106.8 kg
```

Notice `{total_kg:.1f}`. The code `:.1f` means *"show one digit after the decimal point"*. Without it, Python may print a long number that makes your report look messy.

A classic trap: numbers that are really text

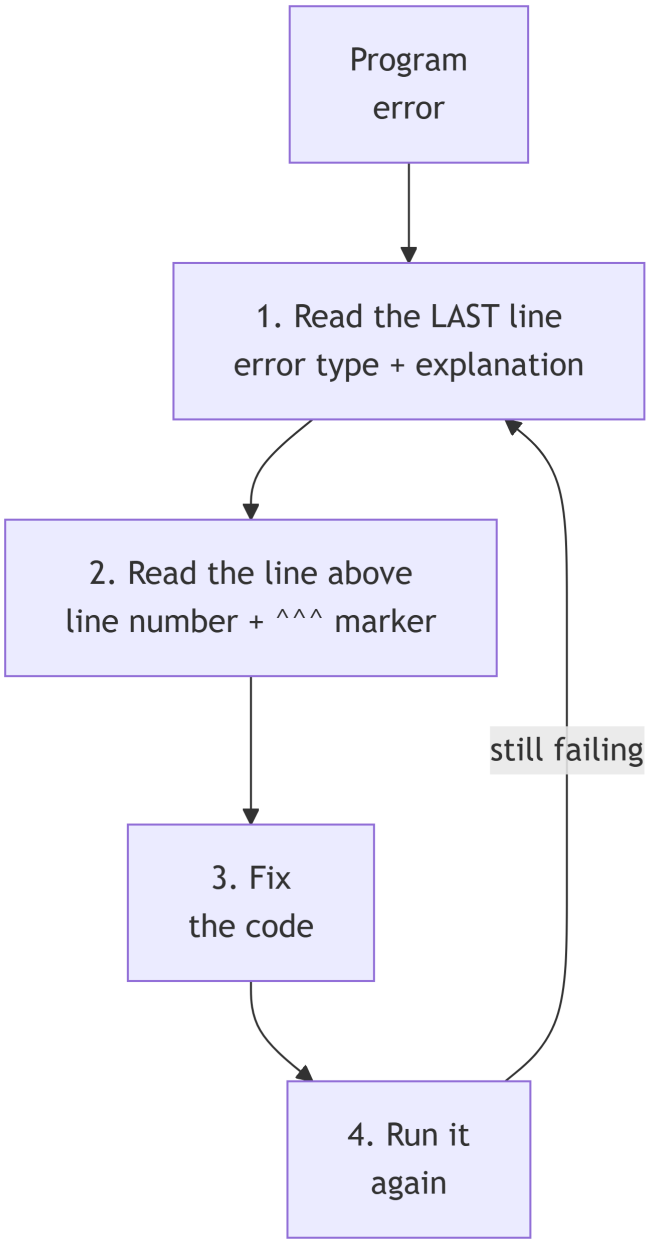
Suppose you ask the user to type a cow's weight, then want to add 10 kg:

```
# Chapter 5 - The trap: input() always returns text (this example fails ON PURPOSE)
weight = input("Cow weight (kg): ")
print(weight + 10)
```

You type `350`, and then… **an error**:

```
Cow weight (kg): 350
Traceback (most recent call last):
  File "04_input_wrong.py", line 3, in <module>
    print(weight + 10)
          ~~~~~~~^~~~
TypeError: can only concatenate str (not "int") to str
```

Relax, your computer isn't broken. Let's read the message slowly, **from the bottom up**:



How to read an error message: start from the last line.

1. **The last line** is the heart of the problem: `TypeError`. It means the **types** don't match. Python can only join text (`str`) to text, not to a number (`int`).
2. **The lines above it** show where the problem is: line 3, right at `weight + 10` (see the `~~~^~~~` marker).

The cause: **`input()` always gives you text**, even when what you typed is a number. So `weight` holds `"350"` (text), not `350` (a number).

The fix is to turn that text into a number with `float()`:

```
# Chapter 5 - The fix: turn text into a number with float()
weight = float(input("Cow weight (kg): "))
print("Target next month:", weight + 10, "kg")
```

```
Cow weight (kg): 350
Target next month: 360.0 kg
```

TECHNICAL STUFF (feel free to skip)

Why does it show `360.0` instead of `360`? Because `float()` produces a decimal number, and decimal numbers are always shown with a dot. If you're sure the weight is always whole, use `int()`. But careful: `int("350.5")` fails, because that text isn't a whole number.

A typo? Python lends a hand

Python 3.14 is quite kind about typos:

```
# Chapter 5 - A typo in a variable name (this example fails ON PURPOSE)
diameter_cm = 32.5
print(diamter_cm)
```

```
Traceback (most recent call last):
  File "06_typo.py", line 3, in <module>
    print(diamter_cm)
          ^^^^^^^^^^
NameError: name 'diamter_cm' is not defined. Did you mean: 'diameter_cm'?
```

`NameError` means Python doesn't recognise that name. The last sentence even guesses what you meant: *"Did you mean: 'diameter_cm'?"*

TIP

Make a habit of reading error messages to the very last line. Python often hands you the answer.

Running Project B2: your first field data

Now let's put it all together. Record one piece of data from each sector:

```
# Chapter 5 - Running project B2: first field data

# Forest
tree_species = "Meranti"
tree_diameter_cm = 45.2

# Fisheries
pond_name = "Pond B"
fish_count = 800

# Livestock
animal_name = "Limousin cattle"
animal_weight_kg = 512.5

print(f"{tree_species} tree: diameter {tree_diameter_cm} cm")
print(f"{pond_name}: {fish_count} fish")
print(f"{animal_name}: {animal_weight_kg} kg")
```

```
Meranti tree: diameter 45.2 cm
Pond B: 800 fish
Limousin cattle: 512.5 kg
```

In Chapter 9, we'll gather these scattered values into a single container (lists and dicts), so they can hold hundreds of trees, ponds, and animals at once.

Exercises

1. Create variables for one rice field: the rice variety (text), area in hectares (decimal), plant age in days (whole number), and whether it has been fertilised (true/false). Print all four with an f-string.
2. Ask the user to type how many eggs were collected today, then print the total for a week (times 7). Don't forget the `input()` trap!
3. **Challenge:** compute the area of a tree crown with a diameter of 6.4 metres (formula: 3.14 × r × r), and display it with two digits after the decimal point.

Chapter Summary

- A **variable** = a name label for a value. `=` means *store into*.
- Four main types: `str` (text), `int` (whole number), `float` (decimal), `bool` (true/false).
- Decimals use a **dot**, not a comma.
- An **f-string** (`f"...{variable}..."`) is the easiest way to print text mixed with numbers.
- `input()` always returns **text**. Convert it with `int()` or `float()` before calculating.
- Read error messages **from the last line**.

Next, in Chapter 6: calculating and comparing. How does Python know whether a cow has reached its target weight?

Chapter 6 — Operators and Input/Output: Calculating, Comparing, and Chatting With Your Program

Series B2 — Python Language Foundations

In this chapter, you will:

- Calculate with Python: add, subtract, multiply, divide, power, and remainder
- Compare values and get a True or False answer
- Combine conditions with `and`, `or`, and `not`
- Control how `print()` looks and receive typing through `input()`
- Build your first fish-pond calculator

All scripts for this chapter are in the `kode/ch06/` folder. On the web version, you can also download them all at once (zip).

In Chapter 5, your program learned to *remember* values. Now it learns three new things: to **calculate**, to **compare**, and to **talk with the user**. The tools for the first two are called **operators**: special signs such as `+` and `>=` that tell Python to do something with values.

Calculating like a calculator

Python can do all the basic sums, and the symbols are nearly the same as on a calculator:

Operator	Meaning	Example	Result
`+`	add	`5 + 3`	`8`
`-`	subtract	`5 - 3`	`2`
`*`	multiply (an asterisk, not `x`)	`5 * 3`	`15`
`/`	divide	`5 / 2`	`2.5`
`//`	whole-number divide	`5 // 2`	`2`
`%`	remainder	`5 % 2`	`1`
`**`	power	`5 ** 2`	`25`

Let's apply them to a fish harvest:

```
# Chapter 6 - Arithmetic: calculating the harvest
harvest_morning_kg = 120.5
harvest_evening_kg = 98.25

total_kg = harvest_morning_kg + harvest_evening_kg
difference_kg = harvest_morning_kg - harvest_evening_kg
price_per_kg = 15000
income = total_kg * price_per_kg

print(f"Total harvest: {total_kg} kg")
print(f"Difference morning vs evening: {difference_kg} kg")
print(f"Income: Rp{income:.0f}")
```

```
plot_side_m = 12
plot_area_m2 = plot_side_m ** 2   # power of two
print(f"Plot area: {plot_area_m2} m2")
```

```
Total harvest: 218.75 kg
Difference morning vs evening: 22.25 kg
Income: Rp3281250
Plot area: 144 m2
```

Notice `{income:.0f}`: the code `:.0f` means "show **no** digits after the point". Without it, the result appears as `3281250.0`.

> TIP
>
> Use the name `plot_area_m2`, not `area`. Putting the unit inside the variable name makes your calculations easy to check: if you accidentally multiply metres by centimetres, the mistake is easy to spot.

Three ways to divide

Division has three forms, and all three are useful in the field. Suppose 1,250 fish are shared equally among 8 ponds:

```
# Chapter 6 - Three kinds of division: /  //  %
fish_count = 1250
pond_count = 8

print(f"Normal division (/):   {fish_count / pond_count}")
print(f"Whole division (//):   {fish_count // pond_count}")
print(f"Remainder (%):         {fish_count % pond_count}")
```

```
Normal division (/):   156.25
Whole division (//):   156
Remainder (%):         2
```

- `/` gives a **decimal** result (156.25 fish per pond). Good for averages.
- `//` gives a **whole-number** result (156 fish per pond). Good for sharing things that can't be cut, like fish.
- `%` gives the **remainder** (2 fish left over). Beginners often forget this one, yet it is very useful.

> REMEMBER
>
> In Python, `/` **always** gives a decimal number (`float`), even when the result is whole. `10 / 2` gives `5.0`, not `5`. For a whole result, use `//`.

One thing you can't do: divide by zero. See what happens when the number of ponds happens to be `0`:

```
# Chapter 6 - A mistake: dividing by zero
fish_count = 1250
pond_count = 0
print(fish_count / pond_count)
```

```
Traceback (most recent call last):
  File "10_divide_by_zero.py", line 4, in <module>
    print(fish_count / pond_count)
          ~~~~~~~~~~~^~~~~~~~~~~~
ZeroDivisionError: division by zero
```

`ZeroDivisionError` means exactly what it says: *division by zero*. In a real program, Chapter 7 will teach how to check `pond_count` before dividing.

Comparing values

Comparison operators compare two values and produce `True` or `False`. Remember the `bool` type from Chapter 5? This is where it gets used.

Operator	Meaning
`==`	equal to
`!=`	not equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to

```
# Chapter 6 - Comparison operators: the result is always True or False
weight_g = 520
target_g = 500

print("weight == target :", weight_g == target_g)
print("weight != target :", weight_g != target_g)
print("weight >  target :", weight_g > target_g)
print("weight >= target :", weight_g >= target_g)
print("weight <  target :", weight_g < target_g)
print("weight <= target :", weight_g <= target_g)

# Text can be compared too, and upper/lower case count as different
species = "Catfish"
print('species == "Catfish" :', species == "Catfish")
print('species == "catfish" :', species == "catfish")
```

```
weight == target : False
weight != target : True
weight >  target : True
weight >= target : True
weight <  target : False
weight <= target : False
species == "Catfish" : True
species == "catfish" : False
```

Two important notes from the results above:

1. Text can be compared too, and **upper and lower case count as different**: `"Catfish"` is not equal to `"catfish"`.
2. A comparison result can be stored in a variable, like `ready_to_harvest = weight_g >= 500`.

⚠ WARNING

This is the most famous trap: **= and == are very different**. One equals sign means *store*, two equals signs mean *compare*. Python helps if you mix them up:

```
# Chapter 6 - A mistake: using = (store) when == (compare) was meant
print(500 = 500)
```

```
  File "09_wrong_equals.py", line 2
    print(500 = 500)
          ^^^^^
SyntaxError: expression cannot contain assignment, perhaps you meant "=="?
```

The message is clear: *"expression cannot contain assignment, perhaps you meant ==?"* The number `500` is not a variable, so it can't be given a value.

Combining conditions: and, or, not

Decisions in the field rarely depend on one condition. Fish are ready to harvest if their weight is enough **and** they are healthy. A pond needs checking if the fish are still young **or** not healthy. Python has three logical operators for this:

Operator	Meaning	Result is `True` when…
`and`	and	**both** sides are `True`
`or`	or	**at least one** side is `True`
`not`	not	its side is `False` (flipped)

```
# Chapter 6 - Logical operators: and, or, not
weight_g = 520
healthy = True
age_days = 90

ready_to_harvest = weight_g >= 500 and healthy
needs_checking = age_days < 30 or not healthy

print("Ready to harvest :", ready_to_harvest)
print("Needs checking   :", needs_checking)
print("Not healthy      :", not healthy)
```

```
Ready to harvest : True
Needs checking   : False
Not healthy      : False
```

Read `ready_to_harvest` like an ordinary sentence: *"weight at least 500 grams and healthy"*. Both conditions hold (520 ≥ 500, and `healthy` is true), so the result is `True`. That is why Python was made to look like human language.

The order operators are carried out

Like in school maths, operators have an order: multiplication is done before addition.

```
# Chapter 6 - The order operators are carried out (like school maths)
print(2 + 3 * 4)       # multiplication is done first
print((2 + 3) * 4)     # brackets change the order
print(10 - 2 ** 3)     # power comes before subtraction
```

```
# Field example: feed cost for 2 ponds, plus an electricity cost
cost = 50000 + 2 * 30000
print(f"Total cost: Rp{cost}")
```

```
14
20
2
Total cost: Rp110000
```

`2 + 3 * 4` gives `14` (not `20`) because `3 * 4` is done first. And `10 - 2 ** 3` gives `2`, because the power `2 ** 3 = 8` is done before the subtraction.

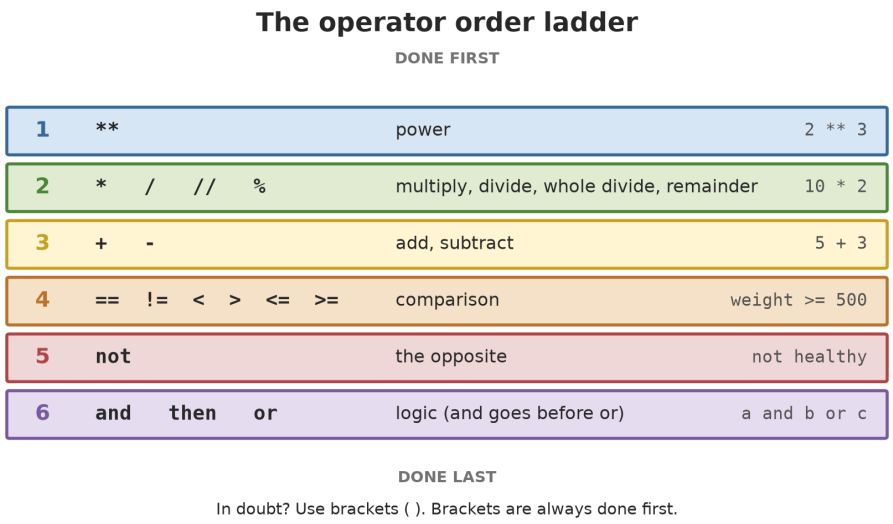


Figure 0.1: The operator order ladder (simplified; operators not covered yet are left out).

> TIP
>
> No need to memorise the ladder. **When in doubt, use brackets.** `(2 + 3) * 4` clearly wants the addition first, and whoever reads your code three months from now doesn't have to guess.

The decimal surprise

This is not your fault, and not Python's fault:

```
# Chapter 6 - A surprise with decimal numbers
print(0.1 + 0.2)
print(0.1 + 0.2 == 0.3)

result = round(0.1 + 0.2, 2)    # round to 2 digits after the point
print(result)
print(result == 0.3)
```

```
0.30000000000000004
False
0.3
True
```

Yes, `0.1 + 0.2` is **not exactly** `0.3` on a computer. The cure is simple: round first with `round(number, decimal_places)` before comparing.

TECHNICAL STUFF (feel free to skip)

Computers store numbers in the binary system (1s and 0s). Some decimal fractions that look simple to us, like `0.1`, cannot be written exactly in binary, just as 1/3 cannot be written exactly in decimal (0.3333...). The result is a tiny difference. This happens in almost every programming language, not only Python.

Talking with the program: print() and input()

You have used `print()` since Chapter 4 and `input()` in Chapter 5. Now let's look at two useful settings of `print()`, and at `input()` more closely.

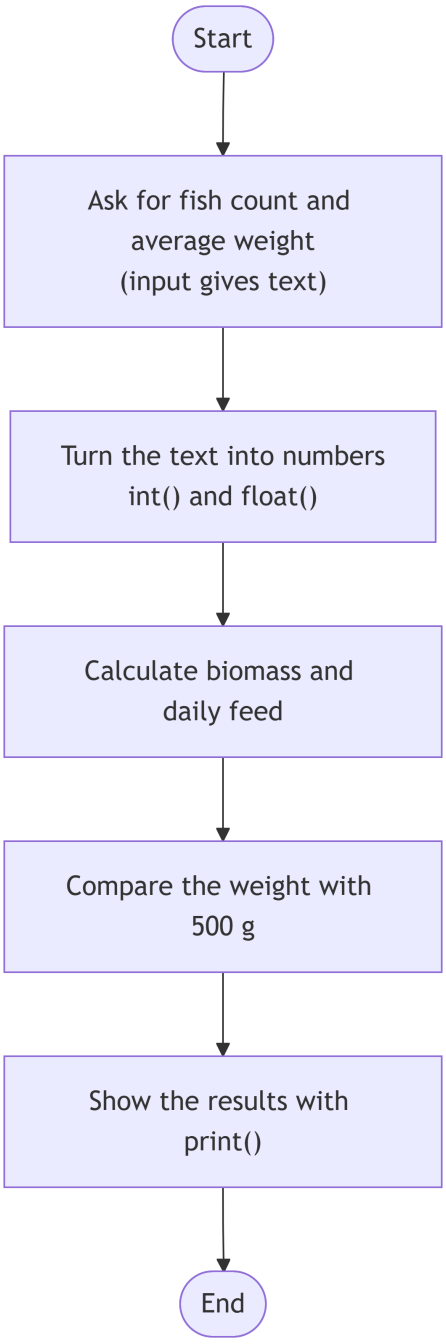
```
# Chapter 6 - print() and input() in more depth
name = input("Business unit name: ")
location = input("Location: ")
print("Business unit:", name, "in", location)

print("Pond", "A", "active", sep="-")     # sep = separator between values
print("Checking...", end=" ")             # end = what closes the line
print("done")
```

```
Business unit name: Mina Tani Pond
Location: Subang
Business unit: Mina Tani Pond in Subang
Pond-A-active
Checking... done
```

- `input("...")` shows a question, waits for you to type, then **hands back the text** you typed.
- In `print()`, **`sep`** sets the separator between values (here a `-` sign), and **`end`** sets what closes the line. Normally `end` is a new line, but here we replace it with one space so "Checking..." and "done" appear on the same line.

Running Project B2: the pond calculator

Now let's combine everything: a program that asks for the number and weight of fish, calculates the biomass and the daily feed need, then decides whether the fish are ready to harvest.



Flow of the pond calculator: ask, turn into numbers, calculate, display.

```
# Chapter 6 - Running project B2: pond calculator
# (the 3% and 500 g figures are practice numbers, not farming advice)
fish_count = int(input("Number of fish: "))
avg_weight_g = float(input("Average weight (grams): "))

biomass_kg = fish_count * avg_weight_g / 1000
daily_feed_kg = biomass_kg * 0.03
ready_to_harvest = avg_weight_g >= 500
```

```
print(f"Pond biomass   : {biomass_kg:.1f} kg")
print(f"Feed per day   : {daily_feed_kg:.2f} kg")
print(f"Ready to harvest: {ready_to_harvest}")
```

```
Number of fish: 800
Average weight (grams): 540.5
Pond biomass   : 432.4 kg
Feed per day   : 12.97 kg
Ready to harvest: True
```

Follow it step by step: 800 fish × 540.5 grams = 432,400 grams = **432.4 kg** of biomass. Daily feed at 3% of that is **12.97 kg**. And since 540.5 ≥ 500, the pond is **ready to harvest**.

> ⚠ WARNING
>
> The 3% feed figure and the 500-gram harvest weight are only **practice numbers**, not farming advice. Real needs depend on the fish species, age, water temperature, and other factors. Ask a fisheries extension officer or an experienced fish farmer.

Exercises

1. A farmer harvests 3 plantation blocks: 245.5 kg, 310 kg, and 198.25 kg. Calculate the total harvest and the average per block, then print them with an f-string.
2. A barn holds 1,000 eggs to be packed in trays of 30. Using `//` and `%`, work out how many full trays there are and how many eggs are left over.
3. Change `08_pond_calculator.py` so the feed percentage (3%) is also asked through `input()`. Don't forget to turn the text into a number and divide by 100.
4. **Challenge:** write one expression that is `True` when a tree is **fit to be felled**: diameter at least 40 cm **and** healthy, **or** height over 25 m. Test it with several numbers. Be careful with the order of `and` and `or`; use brackets.

Chapter Summary

- **Arithmetic:** `+ - * / // % **`. `/` always gives a decimal; `//` divides into whole numbers; `%` gives the remainder.
- **Comparison:** `== != > >= < <=` give `True` or `False`. **`=` stores, `==` compares.**
- **Logic:** `and`, `or`, `not` combine conditions.
- Operators have an **order**; when in doubt, use **brackets**.
- Decimals on a computer can be slightly off; use `round()` before comparing.
- `print()` has `sep` and `end`; `input()` always gives **text**, so convert it with `int()` or `float()`.

Next, in Chapter 7: your program can answer True or False. Now we use that answer to **make decisions**: *if* the weight is enough, *then* harvest; *otherwise*, wait.

Chapter 7 — Conditionals: Teaching a Program to Make Decisions

Series B2 — Python Language Foundations

In this chapter, you will:

- Use `if`, `elif`, and `else` so a program picks its own road
- Understand **indentation**, an important Python rule
- Choose one of many possibilities with `match` and `case`
- Avoid the divide-by-zero error you met in Chapter 6
- Build a fish-pond status checker

All scripts for this chapter are in the `kode/ch07/` folder. On the web version, you can also download them all at once (zip).

In Chapter 6, your program could answer *True* or *False*. But that answer wasn't used for anything yet. In this chapter, the answer becomes a **decision**: *if* the fish are heavy enough, *then* harvest; *otherwise*, wait. This is exactly the **choice** ingredient from Chapter 2.

if: do something only when the condition is true

```
# Chapter 7 - if: do something only when the condition is true
weight_g = 520

if weight_g >= 500:
    print("The fish are ready to harvest")
    print("Write it in the harvest book")

print("Check finished")   # not indented: always carried out
```

```
The fish are ready to harvest
Write it in the harvest book
Check finished
```

Notice three things:

1. After the condition comes a **colon** `:`.
2. The lines that belong to the `if` are **indented** (in this book, 4 spaces). The two indented `print` lines are carried out only when the condition is true.
3. The last line is **not indented**, so it is carried out **always**, whether the condition was true or false.

> **i** REMEMBER
>
> In Python, **indentation is not decoration**: it shows which lines belong to the `if`. Other languages use curly braces; Python uses indentation. Use **4 spaces** consistently.

else: the second road

What if the condition is false? Add `else` ("otherwise"):

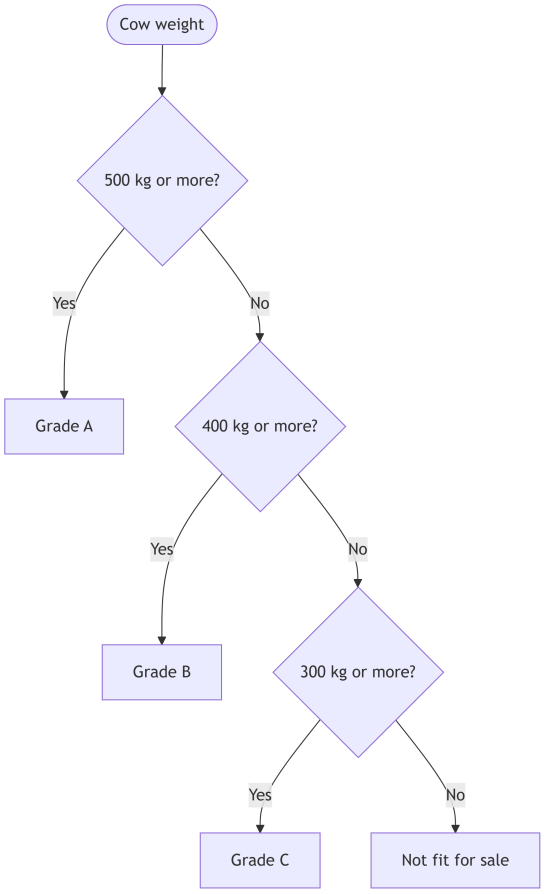
```
# Chapter 7 - if ... else: two roads
water_ph = 5.8

if water_ph < 6:
    print("pH too low: add lime")
else:
    print("pH is fine")
```

```
pH too low: add lime
```

The value `5.8` is less than 6, so the `if` road is chosen. Only **one** of the two roads is carried out, never both.

elif: many roads

Sometimes there are more than two choices. `elif` (short for *else if*, "otherwise, if…") chains the next condition:



Flow for choosing a cow's grade: conditions are checked from the top, stopping at the first true one.

```
# Chapter 7 - if ... elif ... else: many roads (practice numbers)
weight_kg = 450

if weight_kg >= 500:
    grade = "Grade A"
elif weight_kg >= 400:
    grade = "Grade B"
elif weight_kg >= 300:
    grade = "Grade C"
```

```
else:
    grade = "Not fit for sale"

print(f"A {weight_kg} kg cow is: {grade}")
```

```
A 450 kg cow is: Grade B
```

Python checks the conditions **from top to bottom**, and stops at the first true one. Since 450 ≥ 400 but not ≥ 500, the result is Grade B.

> ⚠ WARNING
>
> **The order of conditions matters.** If you wrote `weight_kg >= 300` first, a 450 kg cow would go straight into Grade C, because 450 ≥ 300 is also true. Arrange them from the strictest condition.

> TIP
>
> The grade limits in this example are only **practice numbers**. In a real project, keep limits like these in well-named variables (for example `grade_a_limit_kg`) so they are easy to change in one place.

Combining conditions

The `and`, `or`, and `not` operators from Chapter 6 can be used directly inside an `if`:

```
# Chapter 7 - Combined conditions with and / or
diameter_cm = 45
height_m = 20
healthy = True

if diameter_cm >= 40 and healthy:
    print("The tree is fit to be felled")
elif height_m > 25 or not healthy:
    print("The tree needs checking first")
else:
    print("The tree is left to grow")
```

```
The tree is fit to be felled
```

Trace it: diameter 45 ≥ 40 **and** the tree is healthy, so the first condition is true and the other two are not even checked.

Protecting a program from errors

In Chapter 6 you saw `ZeroDivisionError`. Now we can prevent it: check first before dividing.

```
# Chapter 7 - Avoiding the ZeroDivisionError from Chapter 6
fish_count = 1250
pond_count = 0

if pond_count == 0:
    print("Pond count is 0: division cancelled")
else:
    print(f"Fish per pond: {fish_count // pond_count}")
```

```
Pond count is 0: division cancelled
```

This “check first, then act” pattern will be used often. Chapter 12 teaches another, more general way to handle errors.

match and case: choosing one of many values

When you compare **one variable** against many possible values, a chain of `elif` gets long. Python offers `match` and `case`:

```
# Chapter 7 - match / case: choosing one of many possible values
# (feed amounts are practice numbers, not advice)
animal = "goat"

match animal:
    case "cow":
        feed_kg = 25
    case "goat" | "sheep":
        feed_kg = 4
    case "chicken":
        feed_kg = 0.12
    case _:
        feed_kg = None

if feed_kg is None:
    print("Animal type not registered")
else:
    print(f"Feed for {animal}: {feed_kg} kg per head per day")
```

```
Feed for goat: 4 kg per head per day
```

- `match animal:` means "look at what is in this variable".
- Each `case` is one possibility. The `|` sign means **or** (goat *or* sheep take the same road).
- `case _:` catches **everything else**, like `else`.
- `is None` is the way to compare with `None` (the "empty" value from Chapter 5).

> TECHNICAL STUFF (feel free to skip)
>
> `match`/`case` is available from Python 3.10. It is far more powerful than shown here: it can match the *shape* of data (for example a list holding two numbers). For Series B2, just use it to choose by a single value.

A trap: forgetting to indent

```
# Chapter 7 - A mistake: forgetting to indent after if
weight_g = 520
if weight_g >= 500:
print("The fish are ready to harvest")
```

```
  File "07_wrong_indent.py", line 4
    print("The fish are ready to harvest")
    ^^^^^
IndentationError: expected an indented block after 'if' statement on line 3
```

`IndentationError` means *"I expected an indented line after the `if`"*. The message even names the line number of the `if` that demands the indentation. The cure: move the `print` line inward by 4 spaces.

Running Project B2: pond status

Now we give the pond calculator from Chapter 6 the ability to decide:

```
# Chapter 7 - Running project B2: pond status from average weight
# (the 500 g and 400 g limits are practice numbers)
avg_weight_g = float(input("Average fish weight (grams): "))

if avg_weight_g >= 500:
    status = "READY TO HARVEST"
elif avg_weight_g >= 400:
    status = "ALMOST READY: check again next week"
else:
    status = "STILL GROWING"

print(f"Pond status: {status}")
```

```
Average fish weight (grams): 460
Pond status: ALMOST READY: check again next week
```

With an average weight of 460 grams, the program answers **ALMOST READY**. Try running it with other numbers (for example 520 and 300) to see the three different roads.

Exercises

1. Write a program that asks for the pond water temperature (a decimal number) and prints "Too cold" if below 25, "Too hot" if above 32, and "Good temperature" otherwise. (The limits 25 and 32 are practice numbers.)
2. Change `03_elif.py` so the cow's weight is asked through `input()`. Test with 520, 350, and 200.
3. Make a program that asks for a **plant's age in days** and prints the stage: "Seedling" (0–14), "Vegetative" (15–45), "Flowering" (46–70), or "Harvest" (over 70). These are made-up numbers for practice.
4. **Challenge:** change `06_match_case.py` so `animal` is asked through `input()`, and add one new kind of animal. What happens if the user types `Cow` with a capital C?

Chapter Summary

- **`if`** / **`elif`** / **`else`** make a program pick a road. Only **one** road is carried out.
- A **colon** follows the condition, and the content is **indented** (4 spaces). Indentation is part of Python's grammar.
- Python checks conditions **from top to bottom**; arrange them from the strictest.
- `and`, `or`, `not` combine conditions.
- **`match`** / **`case`** chooses by a single value; `case _` catches the rest.
- Check before dividing, so you don't hit `ZeroDivisionError`.

Next, in Chapter 8: decisions can be made. Now we teach the program to **repeat work**: check every pond, add up the whole harvest, without writing the code dozens of times.

Chapter 8 — Loops: Telling a Program to Do the Same Thing Many Times

Series B2 — Python Language Foundations

In this chapter, you will:

- Use `for` to repeat work on every item in a list
- Make a run of numbers with `range()`
- Use `while` to repeat as long as a condition stays true
- Stop or skip one round with `break` and `continue`
- Build a plantation harvest summary per block

All scripts for this chapter are in the `kode/ch08/` folder. On the web version, you can also download them all at once (zip).

Imagine checking 40 fish ponds. Writing `print` 40 times is obviously tiring, and if the number of ponds grows to 41, you must edit the code again. This is where **loops** save you. It is the third ingredient from Chapter 2.

for: for every item in a list

```
# Chapter 8 - for: repeat for every item in a list
pond_list = ["A", "B", "C"]

for pond in pond_list:
    print(f"Checking Pond {pond}")

print("All ponds have been checked")
```

```
Checking Pond A
Checking Pond B
Checking Pond C
All ponds have been checked
```

Read it like a sentence: *"for every `pond` in `pond_list`, do the indented part."* On the first round `pond` holds `"A"`, on the second `"B"`, and so on. The last line is not indented, so it is carried out only **once**, after all the rounds are finished.

> **i** REMEMBER
>
> The indentation rule from Chapter 7 applies here too: the **indented** lines are the body of the loop that gets repeated; a line that isn't indented is carried out after the loop ends.

range: a run of numbers

If you don't have a list but want to repeat a certain number of times, use `range()`:

```
# Chapter 8 - range(): making a run of numbers
for day in range(1, 6):           # 1, 2, 3, 4, 5 (the number 6 is not included)
    print(f"Day {day}")
```

```
print(list(range(5)))             # starts from 0
print(list(range(0, 10, 3)))      # jumps by 3
```

```
Day 1
Day 2
Day 3
Day 4
Day 5
[0, 1, 2, 3, 4]
[0, 3, 6, 9]
```

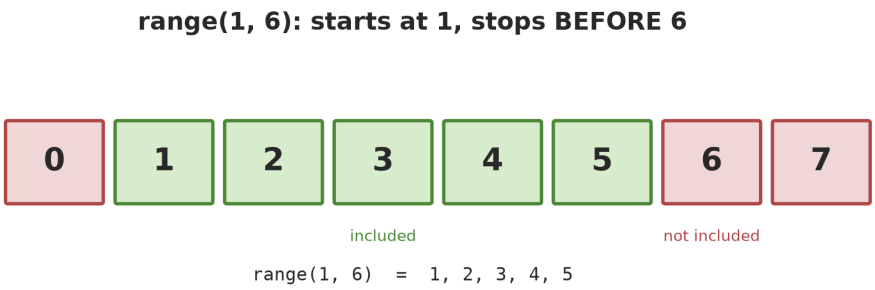


Figure 0.1: `range(1, 6)` stops **before** 6, so it gives 1, 2, 3, 4, 5.

- `range(5)` gives 0, 1, 2, 3, 4: it **starts from 0** and stops before 5.
- `range(1, 6)` gives 1 to 5: the end number is **not included**.
- `range(0, 10, 3)` jumps by 3 each time.
- `list(...)` turns that run into a list so it can be printed.

> ⚠ WARNING
>
> The end number in `range()` is **not included**. This is a classic trap: `range(1, 6)` only reaches 5, not 6. If you need up to 6, write `range(1, 7)`.

The accumulator pattern: adding up with a loop

One of the most useful patterns: prepare a starting variable of 0, then add the list items one by one.

```
# Chapter 8 - Adding up with a loop (the "accumulator" pattern)
daily_harvest_kg = [120.5, 98.25, 143.0]

total_kg = 0
for harvest in daily_harvest_kg:
    total_kg = total_kg + harvest

day_count = len(daily_harvest_kg)      # len() = how many items in the list
print(f"Total harvest: {total_kg} kg in {day_count} days")
print(f"Average: {total_kg / day_count:.2f} kg per day")
```

```
Total harvest: 361.75 kg in 3 days
Average: 120.58 kg per day
```

Trace `total_kg`: it starts at `0`, becomes `120.5`, then `218.75`, then `361.75`. `len(list)` gives **how many items** are in the list, here 3.

TIP

Python actually has `sum(daily_harvest_kg)` which adds everything in one step (Chapter 9). But the accumulator pattern with a loop is important to understand, because the same skeleton is used to count, find the largest, and filter data.

while: repeat while the condition is true

`for` suits situations where you know **how many times** or have **the list**. `while` suits situations where you only know **when to stop**. Remember the feeding flowchart from Chapter 2? Here is its program version:

```
# Chapter 8 - while: repeat as long as the condition stays true
# (follows the feeding flowchart from Chapter 2)
feed_g = 2000
print(f"First scatter: {feed_g} g")

while feed_g < 2400:                    # the fish are still scrambling
    feed_g = feed_g + 200
    print(f"Add 200 g, total {feed_g} g")

print("The fish have calmed down, done")
```

```
First scatter: 2000 g
Add 200 g, total 2200 g
Add 200 g, total 2400 g
The fish have calmed down, done
```

First round: 2000 < 2400, so add 200 to make 2200. Second round: 2200 < 2400, add again to make 2400. Now 2400 < 2400 is **false**, and the loop stops.

WARNING

An endless loop. If you forget to change the variable being checked (for example the line `feed_g = feed_g + 200` is missing), the condition never becomes false and the program spins forever. Don't panic: press **Ctrl + C** in the terminal (Chapter 3) to cancel it.

break and continue

Two helper words to control the rounds:

- **`continue`**: skip the rest of this round, go straight to the next.
- **`break`**: stop completely, leave the loop.

```
# Chapter 8 - continue (skip) and break (stop)
scale_readings_g = [480, 0, 510, 530]

for weight in scale_readings_g:
    if weight == 0:
        print("Scale error, skipped")
        continue
    print(f"Weighing {weight} g")
    if weight >= 500:
        print("Found a fish ready to harvest, stopping")
        break
```

```
Weighing 480 g
Scale error, skipped
```

```
Weighing 510 g
Found a fish ready to harvest, stopping
```

The scale reads `0` (an error), so `continue` skips it. As soon as it finds the 510 g fish that meets the condition, `break` stops the loop: the number 530 is never checked.

A loop inside a loop

```python
# Chapter 8 - A loop inside a loop
for block in ["A", "B"]:
    for row in range(1, 3):
        print(f"Block {block}, row {row}")
```

```
Block A, row 1
Block A, row 2
Block B, row 1
Block B, row 2
```

For **every** block, the inner loop runs **in full**. This pattern is useful for table-shaped data: blocks and rows, ponds and days, plots and months.

A trap: range doesn't accept decimals

```python
# Chapter 8 - A mistake: range() accepts only whole numbers
for i in range(2.5):
    print(i)
```

```
Traceback (most recent call last):
  File "07_range_decimal_wrong.py", line 2, in <module>
    for i in range(2.5):
             ~~~~~^^^^^
TypeError: 'float' object cannot be interpreted as an integer
```

`TypeError` means the data type doesn't fit: `range()` accepts only whole numbers. If you need decimal steps, use `while`.

Running Project B2: plantation harvest summary

Now we combine loops, the accumulator, and decisions (Chapter 7) to summarise the harvest of four plantation blocks:

```python
# Chapter 8 - Running project B2: plantation harvest summary per block
block_names = ["A", "B", "C", "D"]
harvest_kg = [245.5, 310.0, 198.5, 276.0]

total_kg = 0
best_block = block_names[0]
best_harvest = harvest_kg[0]

for i in range(len(block_names)):
    print(f"Block {block_names[i]}: {harvest_kg[i]} kg")
    total_kg = total_kg + harvest_kg[i]
    if harvest_kg[i] > best_harvest:
        best_harvest = harvest_kg[i]
        best_block = block_names[i]

print(f"Total: {total_kg} kg")
```

```
print(f"Average: {total_kg / len(block_names)} kg per block")
print(f"Best block: {best_block} ({best_harvest} kg)")
```

```
Block A: 245.5 kg
Block B: 310.0 kg
Block C: 198.5 kg
Block D: 276.0 kg
Total: 1030.0 kg
Average: 257.5 kg per block
Best block: B (310.0 kg)
```

Three things happen in one loop: printing each block, adding to `total_kg`, and updating the "champion" through an `if`. After the loop ends, the program prints the total, the average (1030.0 ÷ 4 = 257.5), and the best block.

Exercises

1. Print the 7 times table (7 × 1 up to 7 × 10) with `for` and `range()`.
2. Given the cow weights `[350, 412.5, 389.0, 401.5]`, calculate the total and the average with the accumulator pattern.
3. Change `04_while.py` so it adds 150 g each round, starting from 1800 g, stopping when it reaches or passes 2400 g. How many times does it add, and what is the final total?
4. **Challenge:** from the list `[480, 0, 510, 530, 0, 495]`, count the **valid** readings (not 0) and the average of the valid weights, using `continue`.

Chapter Summary

- **`for x in list:`** repeats for every item; **`range(a, b)`** makes a run of numbers and stops **before** `b`.
- **`while condition:`** repeats while the condition is true; make sure something eventually turns it false.
- **`continue`** skips one round; **`break`** stops the whole loop.
- **Accumulator pattern:** start from 0, then add inside the loop.
- The **indented** lines are the body of the loop.
- Stuck in an endless loop? **Ctrl + C**.

Next, in Chapter 9: we have been using lists `[...]` without much explanation. Now we meet the whole family: list, tuple, dict, and set, containers for tree, fish, and livestock data.

Chapter 9 — Built-in Data Structures: Containers for Tree, Fish, and Livestock Data

Series B2 — Python Language Foundations

In this chapter, you will:

- Use a **list** for a list you can change
- Take pieces of a list with **slicing**
- Meet the **tuple** (a fixed list), **dict** (labelled data), and **set** (no duplicates)
- Build a new list in one line with a **comprehension**
- Store the data of several trees at once

All scripts for this chapter are in the `kode/ch09/` folder. On the web version, you can also download them all at once (zip).

Up to Chapter 8, we kept every value in one variable: `weight_g = 520`. But field data is plentiful: hundreds of trees, dozens of ponds. Making hundreds of variables is clearly impossible. Python offers **data structures**: containers that hold many values at once. Picture a filing cabinet with four different kinds of drawers.

Container	Shape	Can it change?	Good for
list	`[800, 1250, 640]`	yes	ordered lists: the fish count of each pond
tuple	`(-6.6, 106.8)`	**no**	fixed data: coordinates
dict	`{"species": "Teak"}`	yes	labelled data: one tree
set	`{"catfish", "tilapia"}`	yes	collections without duplicates: fish species

list: a list you can change

```
# Chapter 9 - list: a list you can change
fish_per_pond = [800, 1250, 640]

print(fish_per_pond)
print("First pond:", fish_per_pond[0])          # numbering starts at 0
print("Last pond:", fish_per_pond[-1])          # -1 = the very last one
print("Number of ponds:", len(fish_per_pond))
print("Total fish:", sum(fish_per_pond))
print("Most crowded pond:", max(fish_per_pond))
```

```
[800, 1250, 640]
First pond: 800
Last pond: 640
Number of ponds: 3
Total fish: 2690
Most crowded pond: 1250
```

Three important points:

- A list is written in **square brackets** `[ ]`, with items separated by commas.
- The position numbers (**indexes**) **start at 0**: the first item is at `[0]`. Negative numbers count from the back: `[-1]` is the last one.
- Useful built-in functions: `len()` (how many), `sum()` (total), `max()` (largest), `min()` (smallest).

A list's contents can be changed after it is made:

```
# Chapter 9 - Changing what is in a list
fish_per_pond = [800, 1250, 640]

fish_per_pond.append(900)           # add at the end
fish_per_pond[0] = 820              # replace item number 0
fish_per_pond.remove(640)           # remove the value 640
print(fish_per_pond)

print(1250 in fish_per_pond)        # is it there?
print(500 in fish_per_pond)

fish_per_pond.sort()                # sort from small to large
print(fish_per_pond)
```

```
[820, 1250, 900]
True
False
[820, 900, 1250]
```

- `append(x)` adds at the **end**; `remove(x)` removes the first matching value.
- `list[0] = 820` replaces item number 0.
- `x in list` answers `True` or `False`: whether the value is there.
- `sort()` sorts the list **in place** (the original list changes).

Slicing: taking a piece

The formula is `list[start:stop:step]`. Like `range()`, the stop number is **not included**.

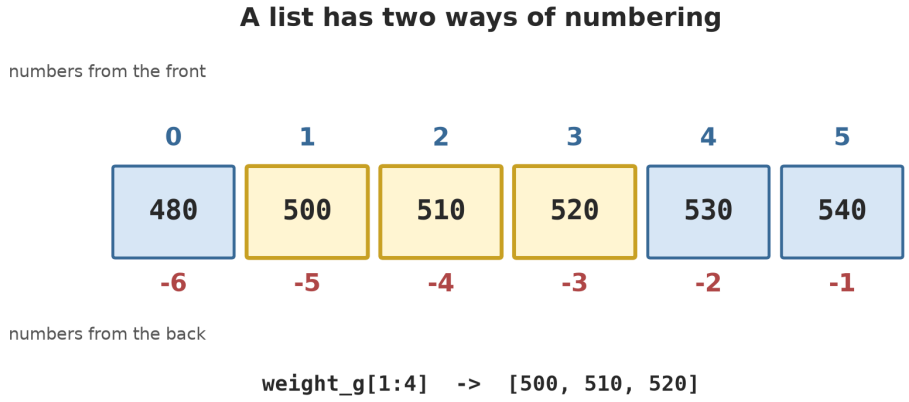


Figure 0.1: A list has two numberings: from the front (starting at 0) and from the back (starting at −1). The slice `[1:4]` takes items 1, 2, 3.

```
# Chapter 9 - Slicing: taking a piece of a list [start:stop:step]
weight_g = [480, 500, 510, 520, 530, 540]

print(weight_g[1:4])     # items 1 to 3 (4 is not included)
print(weight_g[:3])      # the first three
```

```
print(weight_g[3:])      # from item 3 to the end
print(weight_g[::2])     # jump by 2
print(weight_g[::-1])    # reversed
```

```
[500, 510, 520]
[480, 500, 510]
[520, 530, 540]
[480, 510, 530]
[540, 530, 520, 510, 500, 480]
```

A blank part means "from the start" or "to the end". The `[::-1]` trick reverses the whole order.

tuple: a locked list

A tuple resembles a list, but is written with **round brackets** and **cannot be changed** after it is made. It suits data that really shouldn't change, such as location coordinates.

```
# Chapter 9 - tuple: a list that may not be changed (good for coordinates)
pond_location = (-6.6, 106.8)      # (latitude, longitude): sample numbers

latitude, longitude = pond_location    # unpacking a tuple
print(f"Latitude: {latitude}")
print(f"Longitude: {longitude}")
print(pond_location[0])
```

```
Latitude: -6.6
Longitude: 106.8
-6.6
```

The line `latitude, longitude = pond_location` is called **unpacking**: opening a tuple into several variables at once. Trying to change its contents fails:

```
# Chapter 9 - A mistake: trying to change what is inside a tuple
pond_location = (-6.6, 106.8)
pond_location[0] = -6.7
```

```
Traceback (most recent call last):
  File "08_tuple_wrong.py", line 3, in <module>
    pond_location[0] = -6.7
    ~~~~~~~~~~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

The message is clear: *"'tuple' object does not support item assignment"*. That is the whole point of a tuple: protecting data from accidental change.

dict: labelled data

While a list is reached by **number**, a dict is reached by **label** (called a *key*). Perfect for describing one tree:

```
# Chapter 9 - dict: labelled data (key -> value)
tree = {"species": "Teak", "diameter_cm": 32.5, "height_m": 18}

print(tree["species"])
tree["height_m"] = 19              # change a value
tree["healthy"] = True             # add a new label
print(tree)
print(tree.get("colour", "not recorded"))    # safe when the label is missing

for label, content in tree.items():
    print(f"{label}: {content}")
```

```
Teak
{'species': 'Teak', 'diameter_cm': 32.5, 'height_m': 19, 'healthy': True}
not recorded
species: Teak
diameter_cm: 32.5
height_m: 19
healthy: True
```

- Written in **curly brackets** `{ }`, each item as a pair `"key": value`.
- `tree["species"]` fetches the value; `tree["healthy"] = True` adds or changes.
- `.get("colour", "not recorded")` is safe: if the label is missing, it returns a fallback value.
- `.items()` gives label-and-content pairs to use in a `for`.

If the label is missing and you use square brackets, Python complains:

```
# Chapter 9 - A mistake: asking for a label that is not in the dict
tree = {"species": "Teak", "diameter_cm": 32.5}
print(tree["colour"])
```

```
Traceback (most recent call last):
  File "09_missing_label.py", line 3, in <module>
    print(tree["colour"])
          ~~~~^^^^^^^^^^
KeyError: 'colour'
```

`KeyError` means *"key not found"*, and Python names the key that failed.

set: a collection without duplicates

A set automatically drops duplicates and **has no order**. Useful for questions like "which fish species are there?"

```
# Chapter 9 - set: a collection with no duplicates and no order
pond_a_fish = {"catfish", "tilapia", "gourami", "catfish"}
pond_b_fish = {"tilapia", "pangasius"}

print("Species in pond A:", len(pond_a_fish))   # catfish is counted once
print(sorted(pond_a_fish))
print("Tilapia in pond A?", "tilapia" in pond_a_fish)
print("In both ponds:", sorted(pond_a_fish & pond_b_fish))
print("In either pond:", sorted(pond_a_fish | pond_b_fish))
```

```
Species in pond A: 3
['catfish', 'gourami', 'tilapia']
Tilapia in pond A? True
In both ponds: ['tilapia']
In either pond: ['catfish', 'gourami', 'pangasius', 'tilapia']
```

`"catfish"` was written twice but is counted only once. The `&` operator gives the **intersection** (in both ponds) and `|` gives the **union**. We use `sorted()` so the output is in order, because a set itself doesn't promise any order.

Comprehension: a new list in one line

```
# Chapter 9 - Comprehension: making a new list in one line
weight_g = [520, 480, 560, 495]

weight_kg = [g / 1000 for g in weight_g]
ready = [g for g in weight_g if g >= 500]
```

```
print(weight_kg)
print(ready)
```

```
[0.52, 0.48, 0.56, 0.495]
[520, 560]
```

Read `[g / 1000 for g in weight_g]` like a sentence: *"for every `g` in `weight_g`, calculate `g / 1000`."* Adding an `if` at the end filters: `[g for g in weight_g if g >= 500]` takes only those ≥ 500.

> **TIP**
>
> A comprehension is shorthand for a `for` loop that fills a list. If it feels confusing, write it with an ordinary `for` first. Once you understand, then condense it.

Running Project B2: tree data

Now combine: a **list of dicts**. Each tree is one dict, and the plantation is a list of those dicts.

```
# Chapter 9 - Running project B2: tree data as a list of dicts
tree_data = [
    {"species": "Teak", "diameter_cm": 32.5, "height_m": 18, "healthy": True},
    {"species": "Meranti", "diameter_cm": 45.2, "height_m": 27, "healthy": True},
    {"species": "Mahogany", "diameter_cm": 28.0, "height_m": 15, "healthy": False},
]

for tree in tree_data:
    status = "healthy" if tree["healthy"] else "needs checking"
    print(f"{tree['species']:8} diameter {tree['diameter_cm']:5.1f} cm, {status}")

diameters = [tree["diameter_cm"] for tree in tree_data]
print(f"Average diameter: {sum(diameters) / len(diameters):.2f} cm")

healthy_only = [tree["species"] for tree in tree_data if tree["healthy"]]
print("Healthy trees:", healthy_only)
```

```
Teak     diameter  32.5 cm, healthy
Meranti  diameter  45.2 cm, healthy
Mahogany diameter  28.0 cm, needs checking
Average diameter: 35.23 cm
Healthy trees: ['Teak', 'Meranti']
```

A few new things in the `print` line:

- `{tree['species']:8}` means "show with a width of 8 characters", so the columns line up. Single quotes are used inside because the f-string is already wrapped in double quotes.
- `{tree['diameter_cm']:5.1f}` means width 5, one digit after the point.
- `"healthy" if tree["healthy"] else "needs checking"` is a **one-line** `if`: the value is `"healthy"` when true, otherwise the other one.

Average diameter: (32.5 + 45.2 + 28.0) ÷ 3 = **35.23 cm**. The Mahogany is not healthy, so it is left out of the "Healthy trees" list.

Exercises

1. Make a list with the weights of five cows (any numbers). Print the heaviest, the lightest, and the average.
2. From the list `[480, 500, 510, 520, 530, 540]`, use slicing to take: the last three, and the first three reversed.
3. Make a dict for one pond with the labels `name`, `fish_count`, `avg_weight_g`, and `ready_to_harvest` (the result of a comparison). Print everything with `for` and `.items()`.

4. **Challenge:** add one more tree to `10_tree_data_project.py`, then write a list comprehension of the species whose height is more than 20 m.

Chapter Summary

- **list** `[...]`: ordered, changeable; indexes start at **0**, negative from the back.
- **Slicing** `[start:stop:step]`; the stop number isn't included.
- **tuple** `(...)`: unchangeable; **dict** `{key: value}`: labelled data; **set** `{...}`: no duplicates and no order.
- `len()`, `sum()`, `max()`, `min()` work on lists.
- **Comprehensions** build and filter lists in one line.
- A **list of dicts** is the usual way to store many field records.

Next, in Chapter 10: we have data and ways to process it. Now we learn to wrap repeated work into **functions**, so it only needs writing once and can be used many times.

Chapter 10 — Functions: Write Once, Use Many Times

Series B2 — Python Language Foundations

In this chapter, you will:

- Make your own functions with `def`
- Give ingredients in (**arguments**) and receive a result (**return**)
- Understand **scope**: variables inside and outside a function
- Meet the docstring and the lambda
- Finish the field data recorder: the Series B2 project

All scripts for this chapter are in the `kode/ch10/` folder. On the web version, you can also download them all at once (zip).

In Chapter 6, you calculated pond biomass. Imagine you have 12 ponds. Copying the formula 12 times invites mistakes, and if the formula needs fixing, you must fix 12 places. The solution: wrap the formula in a **function**, write it once, call it many times. You have actually been using functions since Chapter 4: `print()`, `input()`, `len()`, and `sum()` are all functions. Now you make your own.

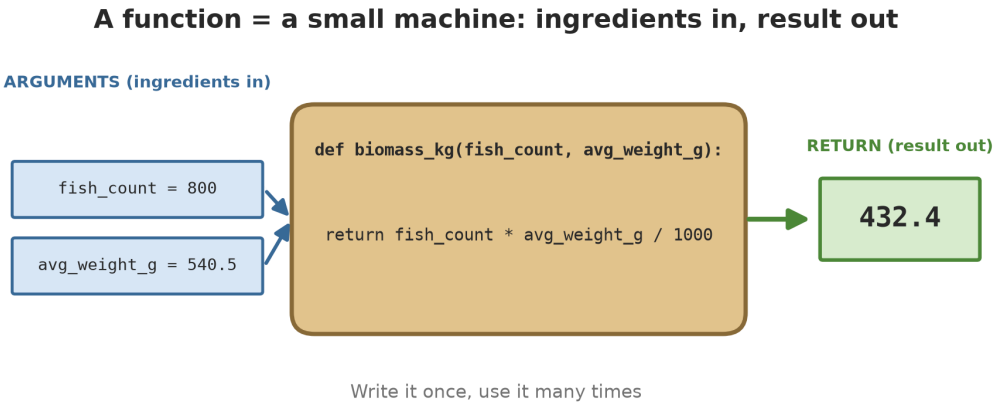


Figure 0.1: A function is like a small machine: ingredients in (arguments), result out (return).

def: making a function

```
# Chapter 10 - def: making your own function
def greet_farmer():
    print("Good morning, farmer!")
    print("May today's harvest be plentiful")


greet_farmer()     # calling the function
greet_farmer()     # it can be called many times
```

```
Good morning, farmer!
May today's harvest be plentiful
```

```
Good morning, farmer!
May today's harvest be plentiful
```

- `def` marks the creation of a function, followed by a name and brackets, then a **colon**.
- The body is **indented** (indentation from Chapter 7).
- A function is only carried out when it is **called**: `greet_farmer()`. Merely making it runs nothing.

> TIP
>
> Name functions with a **verb** that explains the job: `calculate_biomass`, `print_summary`. Whoever reads your code instantly knows what it does without reading the inside.

Arguments and return

A function becomes useful when it takes **ingredients** and gives back a **result**:

```python
# Chapter 10 - Arguments (ingredients in) and return (result out)
def biomass_kg(fish_count, avg_weight_g):
    return fish_count * avg_weight_g / 1000


total = biomass_kg(800, 540.5)
print(f"Pond A biomass: {total:.1f} kg")

# Arguments may be named to make things clearer
print(biomass_kg(fish_count=1250, avg_weight_g=300))
```

```
Pond A biomass: 432.4 kg
375.0
```

- The names inside the brackets (`fish_count`, `avg_weight_g`) are called **parameters**: the places that receive the ingredients. The values you give when calling (`800`, `540.5`) are called **arguments**.
- **`return`** sends the result back to the caller. The result can be stored in a variable (`total`) or used right away.
- Arguments may be named (`fish_count=1250`) to be clearer, and then their order is free.

Default values

A parameter may be given a **default value**, used when the caller doesn't mention it:

```python
# Chapter 10 - Default values for arguments
# (the 3% feed figure is a practice number, not advice)
def daily_feed_kg(biomass, percent=3):
    return biomass * percent / 100


print(f"Using the default percent: {daily_feed_kg(432.4):.2f} kg")
print(f"Percent changed to 2.5:    {daily_feed_kg(432.4, percent=2.5):.2f} kg")
```

```
Using the default percent: 12.97 kg
Percent changed to 2.5:    10.81 kg
```

The first call uses the default `percent=3`. The second replaces it with 2.5. Parameters with default values must be written **after** parameters without them.

Scope: who can see which variable

A variable made **inside** a function is **local**: it lives only while the function runs.

```
# Chapter 10 - Scope: a variable inside a function does not change the one outside
water_temp = 28

def check_temp():
    water_temp = 31                 # a NEW variable, alive only inside the function
    print("Inside the function:", water_temp)


check_temp()
print("Outside the function:", water_temp)
```

```
Inside the function: 31
Outside the function: 28
```

Inside the function, `water_temp = 31` makes a **new** variable that happens to have the same name. The `water_temp` outside stays 28. This protects the program: a function can't accidentally change data outside itself.

> ⚠ WARNING
>
> Because a local variable "dies" after the function finishes, you can't use it from outside:
>
> ```
> # Chapter 10 - A mistake: using a local variable from outside the function
> def calculate():
>     local_result = 10
>
>
> calculate()
> print(local_result)
> ```
>
> ```
> Traceback (most recent call last):
>   File "08_local_variable_wrong.py", line 7, in <module>
>     print(local_result)
>           ^^^^^^^^^^^^
> NameError: name 'local_result' is not defined
> ```
>
> If you need a result from inside a function, **hand it back with `return`**, then catch it outside.

Returning several values

One function can return more than one value at once (really a tuple from Chapter 9), which can be unpacked right away:

```
# Chapter 10 - One function can return several values
def summarise(values):
    return min(values), max(values), sum(values) / len(values)


weight_g = [520, 480, 560, 540]
smallest, largest, average = summarise(weight_g)
print(f"Smallest: {smallest} g")
print(f"Largest: {largest} g")
print(f"Average: {average:.1f} g")
```

```
Smallest: 480 g
Largest: 560 g
Average: 525.0 g
```

Docstring and lambda

A **docstring** is a sentence inside triple quotation marks on the first line of a function. It explains the function's job, and can be read through `__doc__`. A **lambda** is a mini one-line function without a name, often used on the spot, for example as a sorting rule.

```
# Chapter 10 - Docstring (a function's note) and lambda (a mini function)
def plot_area_m2(length_m, width_m):
    """Calculate the area of a rectangular plot in square metres."""
    return length_m * width_m


print(plot_area_m2.__doc__)
print(plot_area_m2(25, 12))

fish = [("catfish", 520), ("tilapia", 480), ("gourami", 610)]
by_weight = sorted(fish, key=lambda item: item[1])    # sort by weight
print(by_weight)
```

```
Calculate the area of a rectangular plot in square metres.
300
[('tilapia', 480), ('catfish', 520), ('gourami', 610)]
```

In `sorted(fish, key=lambda item: item[1])`, the lambda means *"take item number 1 of each pair (the weight) and sort by that"*. The result runs from the lowest weight to the highest.

A classic trap: forgetting return

```
# Chapter 10 - A trap: a function without return gives None
def total_feed(morning_kg, evening_kg):
    morning_kg + evening_kg    # calculated, but not handed back


result = total_feed(2, 3)
print(result)
```

```
None
```

No error, yet the result is **`None`** ("nothing"). The function calculated `morning_kg + evening_kg`, then threw it away because there is no `return`. This is a slippery bug because Python doesn't complain.

> **i** REMEMBER
>
> A function without `return` always gives `None`. If your function's result is suddenly `None`, first check whether you forgot `return`.

Running Project B2: the field data recorder

This is the peak of Series B2: everything you learned works together. There is a **dict** holding **lists** (Chapter 9), two **functions** (this chapter), a **loop** (Chapter 8), and an **f-string** with column formatting (Chapters 5 and 9).

```
# Chapter 10 - Running project B2: field data recorder (summary per sector)
def total(numbers):
    """Add up all the numbers in a list."""
    running = 0
    for n in numbers:
        running = running + n
    return running


def average(numbers):
    """Average of a list of numbers (the list must not be empty)."""
    return total(numbers) / len(numbers)


field_data = {
    "Trees (diameter cm)": [32.5, 45.2, 28.0, 51.5],
    "Plantation (kg)": [245.5, 310.0, 198.5, 276.0],
    "Fish (grams)": [520, 480, 560, 540],
    "Cattle (kg)": [350, 412.5, 389.0],
}

for sector, values in field_data.items():
    print(f"{sector:20} total {total(values):8.1f}   average {average(values):7.2f}")
```

```
Trees (diameter cm)  total    157.2   average   39.30
Plantation (kg)      total   1030.0   average  257.50
Fish (grams)         total   2100.0   average  525.00
Cattle (kg)          total   1151.5   average  383.83
```

Notice that `average()` **calls** `total()`: a function may use another function. Adding a new sector (for example "Eggs (count)") only needs one more line in `field_data`, without changing the functions or the loop. That is the power of functions.

Exercises

1. Write a function `circle_area_m2(radius_m)` that returns the area (use 3.14 for π). Call it for radius 3 and 5.
2. Write a function `harvest_status(weight_g)` that returns `"ready"` if the weight is ≥ 500 and `"not yet"` otherwise. Use it inside a loop over the list `[480, 520, 505, 495]`.
3. Change `09_recorder_project.py`: add the sector "Eggs (count)" with three numbers, and add a function `largest(numbers)` used to print the largest value of each sector. (Don't use `max()`; write the loop yourself.)
4. **Challenge:** write a function `safe_average(numbers)` that returns `None` when the list is empty, so it doesn't hit `ZeroDivisionError`. Use the `if` from Chapter 7.

Chapter Summary

- `def name(parameters):` makes a function; the body is **indented**. A function only runs when **called**.
- **Arguments** = ingredients in; `return` = result out. Without `return`, the result is `None`.
- Parameters may have **default values**.
- Variables inside a function are **local** and invisible from outside.
- A function can return several values, call other functions, and carry a **docstring**.
- **Lambda** = a mini one-line function.

Closing Series B2

Congratulations! You have mastered **all the basic ingredients** of the Python language: variables and data types, operators, decisions, loops, data structures, and functions. With these you can already write a small field data recorder.

But there is one big shortcoming: **the data disappears every time the program closes**. All the numbers in `field_data` must be retyped in the code each time. In Series B3, your program learns to **read and save files**: modules and libraries (Chapter 11), error handling (Chapter 12), and files and data formats (Chapters 13 and 14).

Next, in Chapter 11: modules and packages, the way to borrow ready-made tools from Python's warehouse.

Chapter 11 — Modules and Packages: Borrowing Tools from Python's Warehouse

Series B3 — Small Programs That Are Useful

In this chapter, you will:

- Use `import` to borrow ready-made tools from Python's **standard library**
- Meet the most useful modules: `math`, `random`, `datetime`, `statistics`, `collections`, `itertools`, `pathlib`
- Make your own **module** and understand `if __name__ == "__main__"`
- Build a small **package** with `__init__.py`
- Meet `pip`, the installer for extra libraries, and start the Series B3 project

All scripts for this chapter are in the `kode/ch11/` folder. On the web version, you can also download them all at once (zip).

In Chapter 10, you wrote your own functions. But a farmer doesn't forge a hoe from raw iron every time he wants to dig: he takes it from the tool shed. Python has a big **tool shed**, and every shelf in it is called a **module**: a Python file holding ready-made functions and data. The shed that is installed together with Python is called the **standard library**. You peeked into it in Chapter 4 when you used `import`; now we walk inside.

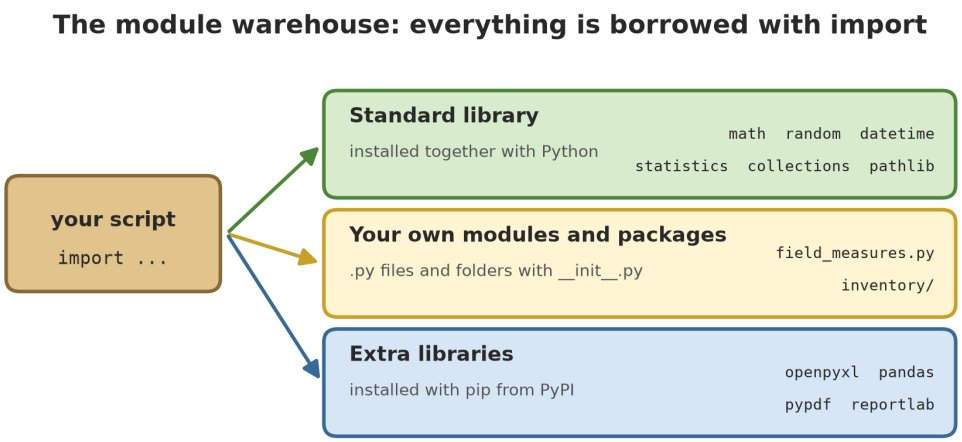


Figure 0.1: The module warehouse: built-in tools, your own tools, and extra tools from pip, all borrowed with import.

Three ways to use import

```
# Chapter 11 - Three ways to use import
import math                          # way 1: bring the whole module, call with the
prefix math.
from math import sqrt              # way 2: take just one tool
import statistics as stat          # way 3: give it a nickname (alias)

print("pi          =", math.pi)
print("sqrt of 144 =", sqrt(144))
```

```
print("mean        =", stat.mean([32.5, 45.2, 28.0]))

# Crown area of a tree, a circle with a radius of 3.5 m
area = math.pi * 3.5 ** 2
print(f"Crown area  = {area:.2f} m2")
```

```
pi          = 3.141592653589793
sqrt of 144 = 12.0
mean        = 35.233333333333334
Crown area  = 38.48 m2
```

- `import math` brings in the whole module; its tools are called with a prefix: `math.pi`.
- `from math import sqrt` takes **one tool** only, used without a prefix.
- `import statistics as stat` gives it a **nickname** (alias) to make it shorter.

The prefix is useful: `math.pi` tells you at once that `pi` comes from the `math` module, not from a variable of your own.

> TIP
>
> Write all `import` lines at the **very top** of the script. A reader instantly knows which tools the program needs.

math and random

```
# Chapter 11 - math and random (with a seed so the result can be repeated)
import math
import random

print(math.ceil(4.2), math.floor(4.8), round(4.5), round(5.5))   # rounding

random.seed(42)                                          # seed: the "random" sequence
becomes fixed
chosen_plot = random.choice(["A", "B", "C"])
sample = random.sample(range(1, 101), 5)        # 5 different tree numbers from 1..100
weather = random.randint(24, 33)                 # whole number 24..33
print("Chosen plot :", chosen_plot)
print("Sample trees:", sample)
print("Fake temp   :", weather)

random.seed(42)                                    # same seed -> same result
print("Again      :", random.choice(["A", "B", "C"]), random.sample(range(1, 101), 5))
```

```
5 4 4 6
Chosen plot : C
Sample trees: [15, 4, 95, 36, 32]
Fake temp   : 27
Again       : C [15, 4, 95, 36, 32]
```

The first line holds a surprise: `round(4.5)` gives **4**, but `round(5.5)` gives **6**. Python rounds a number sitting exactly in the middle to the nearest **even** number, not always up. This rule is called banker's rounding; just remember not to be surprised when it happens.

The `random` module produces "random" numbers. With `random.seed(42)` we set the **seed**: the starting point of the random sequence. Same seed, same sequence; see the last line, which repeats `C [15, 4, 95, 36, 32]`. This is how you make fake data that can be repeated exactly. One honest note: the Python documentation only guarantees that the `random.random()` sequence stays the same for the same seed in future Python versions, while `choice` and `sample` may change between versions. So the numbers above

will be identical on your computer if the Python version is the same, but do not build a system that depends on them.

Dates with datetime

```
# Chapter 11 - datetime: planting date, age, and expected harvest
from datetime import date, timedelta

planted = date(2026, 3, 1)
harvest = planted + timedelta(days=120)      # sample rice: 120 days
age = date(2026, 5, 15) - planted            # difference of two dates = timedelta

print("Planted:", planted)
print("Harvest:", harvest.strftime("%d-%m-%Y"))
print("Age on 15 May 2026:", age.days, "days")
print("From text:", date.fromisoformat("2026-07-01"))

day_names = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
"Sunday"]
print("Harvest day falls on:", day_names[harvest.weekday()])
```

```
Planted: 2026-03-01
Harvest: 29-06-2026
Age on 15 May 2026: 75 days
From text: 2026-07-01
Harvest day falls on: Monday
```

`date(2026, 3, 1)` makes a date. Adding `timedelta(days=120)` shifts the date. The difference between two dates is a `timedelta`, and `.days` takes its number of days (75 days from 1 March to 15 May). `strftime("%d-%m-%Y")` controls how the date is displayed. (We deliberately avoid `date.today()` because its result changes every day.)

statistics and collections

```
# Chapter 11 - statistics: summarising numbers without writing the formulas yourself
import statistics

weight_g = [520, 480, 560, 540, 480, 505]     # sample fish weights (sample numbers)
print("mean     :", round(statistics.mean(weight_g), 2))
print("median   :", statistics.median(weight_g))
print("mode     :", statistics.mode(weight_g))
print("std dev  :", round(statistics.stdev(weight_g), 2))
print("smallest/largest:", min(weight_g), max(weight_g))
```

```
mean     : 514.17
median   : 512.5
mode     : 480
std dev  : 32.31
smallest/largest: 480 560
```

Those six fish weights give a mean of 514.17 g and a median of 512.5 g (the middle value after sorting). The mode, 480, is the number that appears most often. `stdev` (standard deviation) measures how spread out the numbers are; the bigger it is, the less uniform the fish.

```
# Chapter 11 - collections: Counter and defaultdict
from collections import Counter, defaultdict

species = ["Teak", "Meranti", "Teak", "Mahogany", "Teak", "Meranti"]
```

```
count = Counter(species)
print(count)
print("Most common:", count.most_common(1))

per_plot = defaultdict(list)                    # a new key automatically gets an
empty list
for plot, diameter in [("A", 32.5), ("B", 28.0), ("A", 45.2), ("B", 51.5)]:
    per_plot[plot].append(diameter)
print(dict(per_plot))
```

```
Counter({'Teak': 3, 'Meranti': 2, 'Mahogany': 1})
Most common: [('Teak', 3)]
{'A': [32.5, 45.2], 'B': [28.0, 51.5]}
```

`Counter` counts how many times each value appears: Teak appears 3 times. `defaultdict(list)` is a dict that automatically creates an empty list for a new key, so you can `append` right away without first checking whether the key exists.

itertools and pathlib

```
# Chapter 11 - itertools: combinations and groups
from itertools import combinations, product, groupby

ponds = ["A", "B", "C"]
print("Pond pairs :", list(combinations(ponds, 2)))
print("Pond x feed:", list(product(ponds[:2], ["pellet", "chopped"])))

trees = [("A", "Teak"), ("A", "Meranti"), ("B", "Mahogany"), ("B", "Teak")]
for plot, members in groupby(trees, key=lambda t: t[0]):    # data must already be
in order
    print(plot, [species for _, species in members])
```

```
Pond pairs : [('A', 'B'), ('A', 'C'), ('B', 'C')]
Pond x feed: [('A', 'pellet'), ('A', 'chopped'), ('B', 'pellet'), ('B', 'chopped')]
A ['Teak', 'Meranti']
B ['Mahogany', 'Teak']
```

`combinations` makes all pairs, `product` makes all combinations of two lists, and `groupby` groups data that is **already in order**. Note: `groupby` only joins neighbouring elements, so sort the data first if it isn't sorted.

```
# Chapter 11 - pathlib: a file address as an object (does not touch the disk yet)
from pathlib import Path

file = Path("data") / "trees.csv"            # the / sign joins parts of an address
print(file)
print("name       :", file.name)
print("suffix     :", file.suffix)
print("without suffix:", file.stem)
print("folder     :", file.parent)
print("change suffix:", file.with_suffix(".json"))
print("exists on disk?", file.exists())
print("this script is called:", Path(__file__).name)
```

```
data\trees.csv
name       : trees.csv
suffix     : .csv
without suffix: trees
folder     : data
change suffix: data\trees.json
```

```
exists on disk? False
this script is called: 07_pathlib_intro.py
```

`pathlib.Path` treats a file address as an **object**. The `/` sign joins parts of an address. On a Windows computer the separator is displayed as `\`; on macOS or Linux it is displayed as `/`. That is the benefit of `Path`: you write once, and Python adapts to the system. In Chapter 13 we will use it to read and write real files.

Making your own module

A **module** is simply a `.py` file. Our module lives in the `module/` subfolder:

```
# module/field_measures.py (its first two functions)
def basal_area_m2(diameter_cm): ...
def biomass_kg(fish_count, avg_weight_g): ...
```

The full content is in the file `kode/ch11/module/field_measures.py`. The following script uses it:

```
# Chapter 11 - Using our own module from the "module" subfolder
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent / "module"))    # tell Python where the
module lives

import field_measures as measures

print(f"Basal area of 32.5 cm: {measures.basal_area_m2(32.5):.4f} m2")
print(f"Biomass of pond A    : {measures.biomass_kg(800, 540.5):.1f} kg")
print("Name of this module:", measures.__name__)
```

```
Basal area of 32.5 cm: 0.0830 m2
Biomass of pond A    : 432.4 kg
Name of this module: field_measures
```

Basal area is the cross-section area of a trunk; for a diameter of 32.5 cm the result is 0.0830 m². Python looks for modules in a number of folders; we add the `module/` folder to its list through `sys.path.insert`. The module name (`field_measures`) is the file name without `.py`.

`if __name__ == "__main__"`

Every module has a built-in variable `__name__`. When a file is **run directly**, `__name__` holds `"__main__"`. When a file is **imported**, `__name__` holds the module's name. That way one file can have a "quick test" that runs only when it is run directly:

```
# Chapter 11 - if __name__ == "__main__": run directly or imported
import subprocess
import sys
from pathlib import Path

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

import name_demo                                  # imported: __name__ = the name of
the module

print("--- now run directly ---")
result = subprocess.run([sys.executable, str(module / "name_demo.py")],
                        capture_output=True, text=True)
print(result.stdout.strip())
```

```
print("--- quick test in module field_measures (run directly) ---")
result = subprocess.run([sys.executable, str(module / "field_measures.py")],
                        capture_output=True, text=True)
print(result.stdout.strip())
```

```
Inside name_demo, __name__ = name_demo
--- now run directly ---
Inside name_demo, __name__ = __main__
--- quick test in module field_measures (run directly) ---
Quick test: 0.083 432.4
```

The first line appears when `name_demo` is imported (the module's name is `name_demo`). After that the script runs the same file as a program of its own, and this time `__name__` holds `__main__`. In `field_measures.py`, the part under `if __name__ == "__main__":` appears only on the second call; when it is imported in script 08, that quick test does **not** get in the way.

> ⚠ WARNING
>
> Code at the outermost level of a module is **run when it is imported**. So don't put long commands or `print` there, except under `if __name__ == "__main__":`.

Packages: a folder of modules

When modules multiply, gather them in a **package**: a folder holding a file named `__init__.py` plus the modules. Our package is called `inventory`:

```
inventory/
    __init__.py      <- marks the package; may list the tools it shows off
    core_data.py     <- sample data for five sectors
    calc.py          <- basal_area_m2, biomass_kg
    summarize.py     <- summarize(numbers)
```

```
# Chapter 11 - A small package: the "inventory" folder holds __init__.py and its modules
import inventory
from inventory import biomass_kg             # available through __init__.py
from inventory.core_data import fish           # take directly from a module inside
the package

print("Package contents (__all__):", inventory.__all__)
print("The calc module comes from:", inventory.calc.__name__)

for pond in fish:
    kg = biomass_kg(pond["count"], pond["avg_weight_g"])
    print(f"Pond {pond['pond']}: {kg:.1f} kg")
```

```
Package contents (__all__): ['basal_area_m2', 'biomass_kg', 'summarize']
The calc module comes from: inventory.calc
Pond A: 432.4 kg
Pond B: 375.0 kg
Pond C: 288.0 kg
```

`__init__.py` imports tools from its modules so users only need to write `from inventory import biomass_kg`. Pond A holds 800 fish of 540.5 g: 432.4 kg, exactly as in Chapter 10. The `__all__` variable lists the names the package deliberately shows.

pip: borrowing from outside the warehouse

When the tool you need is not in the standard library, **pip** installs it from **PyPI**, the online store of the Python community. Its commands are typed in the terminal, not inside a script. The commands below are only examples and were **not run** when this book was built, because installing packages changes the reader's computer:

```
python -m pip install openpyxl     # installs the Excel library (used in Chapter 15)
python -m pip list                 # shows installed libraries
python -m pip uninstall openpyxl   # removes it
```

Writing `python -m pip` (instead of just `pip`) makes sure the pip you use belongs to the same Python that runs your scripts. If you try to import a library that isn't installed, this is what happens:

```
# Chapter 11 - Pitfall: importing a module that does not exist
import library_does_not_exist

print("this line is never reached")
```

```
Traceback (most recent call last):
  File "11_module_not_found.py", line 2, in <module>
    import library_does_not_exist
ModuleNotFoundError: No module named 'library_does_not_exist'
```

The `ModuleNotFoundError` message means Python could not find a module with that name: either a typo, or it hasn't been installed with pip. Full coverage of installing and separate environments (venv) is in Chapter 17.

> TECHNICAL STUFF (feel free to skip)
>
> When you write `import x`, Python looks for `x` in order: modules already built in, the folder where the script runs, then other folders on `sys.path`. Because the script's folder comes early, you should **not name your own files** `random.py` or `statistics.py`: your module would hide the real one and cause confusing errors.

Running Project B3: summary of five sectors

Welcome to the Series B3 project. In Series B2, the field data lived inside the code. In this series we will save it to files, make it resistant to mistakes, and turn it into reports. Step one: pack our tools into the `inventory` package (which you have already seen), then use it to summarise four tables of data from five sectors (forest, orchard, field crops, fisheries, livestock). The numbers are fictional.

```
# Chapter 11 - Running project B3: summary of five sectors using our own package
from inventory import summarize
from inventory.core_data import trees, crops, fish, livestock

sectors = {
    "Forest (diameter cm)": [t["diameter_cm"] for t in trees],
    "Orchard+field (kg)": [c["yield_kg"] for c in crops],
    "Fisheries (g/fish)": [f["avg_weight_g"] for f in fish],
    "Livestock (kg)": [x["weight_kg"] for x in livestock],
}

for name, numbers in sectors.items():
    r = summarize(numbers)
    print(f"{name:22} n={r['count']}  total={r['total']:8.1f}  mean={r['mean']:7.2f}
median={r['median']}")
```

```
Forest (diameter cm)   n=6  total=   234.0  mean=  39.00  median=38.4
Orchard+field (kg)     n=4  total=  1030.0  mean= 257.50  median=260.75
Fisheries (g/fish)     n=3  total=  1320.5  mean= 440.17  median=480.0
Livestock (kg)         n=4  total=  1190.0  mean= 297.50  median=369.5
```

Read each line like this: the forest sector has 6 trees, a total diameter of 234.0 cm, a mean of 39.00 cm, and a median of 38.4 cm. Orchard and field crops (rice, maize, coffee, cocoa) yield 1030.0 kg. The mean fish weight per pond is 440.17 g, and livestock averages 297.50 kg. Notice how short this script is: all the heavy work is done by the package.

Exercises

1. Use `datetime`: work out the harvest date for coffee planted on 15 January 2026 and harvested 270 days later, then print it in `dd-mm-yyyy` format.
2. With `random.seed(7)`, choose 3 different pond numbers from 1 to 12 using `random.sample`. Run it twice and make sure the result is the same.
3. Add a function `circle_circumference_m(radius_m)` to `module/field_measures.py`, then call it from a new script that imports it.
4. **Challenge:** add a module `inventory/dates.py` with a function `planting_date(harvest_date, age_days)`, export it through `__init__.py`, then use it to print the planting date of each crop in `core_data.crops` if all were harvested on 1 October 2026.

Chapter Summary

- A **module** = a Python file holding tools; a **package** = a folder of modules with `__init__.py`.
- Three forms of import: `import x`, `from x import y`, `import x as z`.
- Useful standard library modules: `math`, `random` (use `seed`), `datetime`, `statistics`, `collections`, `itertools`, `pathlib`.
- `if __name__ == "__main__":` separates code that runs only when the file is run directly.
- `python -m pip install <name>` installs extra libraries; `ModuleNotFoundError` means the module was not found.
- Project B3 has begun: the `inventory` package summarises data from five sectors.

Next, in Chapter 12: error handling, how to keep a program from collapsing the moment something goes wrong.

Chapter 12 — Error Handling: Programs That Don't Collapse Easily

Series B3 — Small Programs That Are Useful

In this chapter, you will:

- Catch errors with `try`, `except`, `else`, and `finally`
- Recognise the error types that appear most often
- Raise your own errors with `raise` and make your own error types
- Read a **traceback** (the trail of an error) from the bottom up
- Hunt bugs with `print`, `assert`, `logging`, and meet the `pdb` debugger

All scripts for this chapter are in the `kode/ch12/` folder. On the web version, you can also download them all at once (zip).

Imagine a field officer typing "thirty" when the computer is waiting for the number 30. Without preparation, the program stops abruptly with a scary red message. In earlier chapters, errors arrived as lessons. Now you learn to prepare for them, like a farmer putting a fence around a pond: not hoping nobody stumbles, but making sure nobody falls in.

In Python, an error is called an **exception**. When one happens and nobody catches it, the program stops. Our job: catch the ones that can be fixed, and let the truly fatal ones show up clearly.

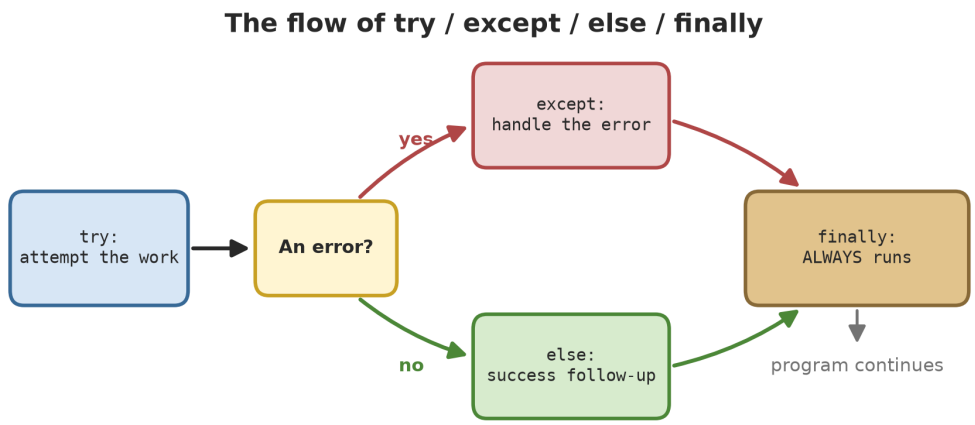


Figure 0.1: The try flow: the try part is attempted; an error goes to except, no error goes to else; finally always runs.

try and except

```
# Chapter 12 - try and except: catching errors so the program does not collapse
diameter_text = "thirty"            # the officer typed a word, not a number

try:
    diameter = float(diameter_text)
    print("Diameter:", diameter)
except ValueError:
    print("Diameter must be a number, not:", diameter_text)
```

```
print("The program keeps running to the last line")
```

```
Diameter must be a number, not: thirty
The program keeps running to the last line
```

Python tries the content of `try`. When `float("thirty")` fails with a `ValueError`, it jumps to `except ValueError`. The rest of the `try` body is skipped, but the program **carries on**: the last line is still printed.

> ⚠ WARNING
>
> Don't write an empty `except:` or catch everything with `except Exception:` and then stay silent. Real errors get hidden and you won't know why the result looks odd. Catch the error type **you expect**, and do something meaningful (give a message, log it, or use a fallback value).

Know your error types

Every error has a **type** and a **message**. The following script deliberately causes eight common errors and shows them:

```
# Chapter 12 - The error types that appear most often
def attempt(name, action):
    try:
        action()
    except Exception as e:
        print(f"{name:14} -> {type(e).__name__}: {e}")


items = [10, 20, 30]
data = {"species": "Teak"}

attempt("ValueError", lambda: int("three"))
attempt("ZeroDivision", lambda: 10 / 0)
attempt("IndexError", lambda: items[5])
attempt("KeyError", lambda: data["diameter"])
attempt("TypeError", lambda: "age: " + 5)
attempt("NameError", lambda: missing_name)
attempt("AttributeError", lambda: items.add(40))
attempt("FileNotFound", lambda: open("does_not_exist.txt"))
```

```
ValueError     -> ValueError: invalid literal for int() with base 10: 'three'
ZeroDivision   -> ZeroDivisionError: division by zero
IndexError     -> IndexError: list index out of range
KeyError       -> KeyError: 'diameter'
TypeError      -> TypeError: can only concatenate str (not "int") to str
NameError      -> NameError: name 'missing_name' is not defined
AttributeError -> AttributeError: 'list' object has no attribute 'add'
FileNotFound      ->  FileNotFoundError:  [Errno  2]  No  such  file  or  directory:
'does_not_exist.txt'
```

Type	Meaning	Field example
`ValueError`	wrong value for that type	turning "three" into a number
`ZeroDivisionError`	divided by zero	average of an empty list
`IndexError`	position outside the list	taking plot no. 6 when there are 3

Type	Meaning	Field example
`KeyError`	dict key does not exist	asking for a “diameter” column that isn’t there
`TypeError`	mismatched types	joining text with a number
`NameError`	variable name does not exist yet	typo in a variable name
`AttributeError`	object lacks that feature	calling `.add()` on a list
`FileNotFoundError`	file not found	wrong file name (see Chapter 13)

In that script, `except Exception as e` is used **only as a demonstration tool**, so all types can be displayed with `type(e).__name__` (the type name) and `e` (the message).

else and finally

```
# Chapter 12 - try / except / else / finally
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("  [except] divisor is zero")
        return None
    else:
        print("  [else] no error")
        return result
    finally:
        print("  [finally] always runs")


print("divide(10, 4):")
print("  result =", divide(10, 4))
print("divide(10, 0):")
print("  result =", divide(10, 0))
```

```
divide(10, 4):
  [else] no error
  [finally] always runs
  result = 2.5
divide(10, 0):
  [except] divisor is zero
  [finally] always runs
  result = None
```

- **`else`** runs **only when there was no error** in `try`.
- **`finally`** runs **always**, error or not, even when there is a `return`. Good for “cleaning up”, such as closing files.

Look at divide(10, 4): the order is `else` then `finally`. With divide(10, 0): `except` then `finally`.

raise: raising your own error

Sometimes there is no built-in error, yet the data is clearly nonsense. Water pH can only be 0 to 14. We raise the error ourselves with `raise`:

```
# Chapter 12 - raise: we raise the error ourselves
def check_ph(ph):
    if not 0 <= ph <= 14:
```

```
        raise ValueError(f"pH {ph} is outside the range 0-14")
    return ph


for value in [7.2, 15, -1]:
    try:
        print("Valid pH:", check_ph(value))
    except ValueError as e:
        print("Rejected:", e)
```

```
Valid pH: 7.2
Rejected: pH 15 is outside the range 0-14
Rejected: pH -1 is outside the range 0-14
```

pH 7.2 passes; 15 and −1 are rejected with the message we wrote. A clear error message saves a lot of time: say **what** is wrong and **its value**.

Your own error types

In a larger project, make your own error types by inheriting from `Exception` (**classes** and inheritance are covered fully in Chapter 16; here just follow the pattern):

```
# Chapter 12 - Making our own error types
class FieldDataError(Exception):
    """Parent of all our field data errors."""


class UnreasonableDiameter(FieldDataError):
    def __init__(self, value):
        super().__init__(f"diameter {value} cm is unreasonable (must be 1 to 300)")
        self.value = value


def check_diameter(cm):
    if not 1 <= cm <= 300:
        raise UnreasonableDiameter(cm)
    return cm


for cm in [32.5, 0.2, 450]:
    try:
        print("OK:", check_diameter(cm))
    except FieldDataError as e:            # catches every descendant of FieldDataError
        print(f"{type(e).__name__}: {e} (value={e.value})")
```

```
OK: 32.5
UnreasonableDiameter: diameter 0.2 cm is unreasonable (must be 1 to 300) (value=0.2)
UnreasonableDiameter: diameter 450 cm is unreasonable (must be 1 to 300) (value=450)
```

`UnreasonableDiameter` is a **child** of `FieldDataError`, so `except FieldDataError` catches both. Handy: one `except` can handle the whole family of our project's errors, while other built-in errors pass through.

Reading a traceback

When an error is not caught, Python prints a **traceback**: the trail of function calls leading to the error.

```
# Chapter 12 - Reading a traceback: an error three functions deep
def average(numbers):
    return sum(numbers) / len(numbers)
```

```
def summarize_plot(name, diameters):
    avg = average(diameters)
    print(f"Plot {name}: average {avg:.1f} cm")


def report():
    summarize_plot("A", [32.5, 45.2])
    summarize_plot("B", [])         # plot B has no data yet


report()
```

```
Plot A: average 38.9 cm
Traceback (most recent call last):
  File "06_read_traceback.py", line 16, in <module>
    report()
    ~~~~~~^^
  File "06_read_traceback.py", line 13, in report
    summarize_plot("B", [])         # plot B has no data yet
    ~~~~~~~~~~~~~~~^^^^^^^^
  File "06_read_traceback.py", line 7, in summarize_plot
    avg = average(diameters)
  File "06_read_traceback.py", line 3, in average
    return sum(numbers) / len(numbers)
           ~~~~~~~~~~~~~^~~~~~~~~~~~~~
ZeroDivisionError: division by zero
```

Read it from the bottom up:

1. **The last line** is the heart of it: `ZeroDivisionError: division by zero`, the error type and message.
2. The line `File ... in average` just above shows the **scene of the incident**: the division `sum(numbers) / len(numbers)` where `len(numbers)` is 0.
3. Further up, `summarize_plot` called it, then `report`, then `<module>` (the outermost code). That is the chain of calls.
4. The `~~~^^^` marks under a line of code point to the troubled part of the expression.
5. The real cause is in the data: plot B has an empty list. Plot A was printed before the error, so the trail also tells you **how far** the program got.

> TIP
>
> Line numbers in a traceback are counted from the first line of the file, **including the comment line at the very top**. That is why the numbers shown above match exactly the lines in the book's code and in the downloadable file: count the opening comment as line 1.

Hunting bugs: print, assert, logging

A program mistake is called a **bug**. Hunting it is called **debugging**. Three basic tools:

1. print debugging: print the variables at each step. Use `repr()` so hidden quote marks and spaces become visible.

```
# Chapter 12 - Debugging with print: look inside variables at every step
def total_weight_kg(ponds):
    total = 0
    for p in ponds:
        weight_kg = p["count"] * p["weight_g"] / 1000
          print("  [debug]", repr(p["name"]), "->", weight_kg)      # repr shows the
quote marks
        total += weight_kg
```

```
    return total


ponds = [
    {"name": "A", "count": 800, "weight_g": 540.5},
    {"name": "B", "count": 1250, "weight_g": 300.0},
]
print("Total:", total_weight_kg(ponds), "kg")
```

```
  [debug] 'A' -> 432.4
  [debug] 'B' -> 375.0
Total: 807.4 kg
```

Pond A is 432.4 kg and pond B 375.0 kg, adding up to 807.4 kg. With the `[debug]` lines you see each term of the sum. Delete those lines once the bug is found.

2. assert: a safety rail that states "this is certainly true". If it is false, the program stops with an `AssertionError`.

```
# Chapter 12 - assert: a safety rail saying "this is CERTAINLY true" (the passing
version)
def survival_percent(start, alive):
    assert start > 0, "the starting count must be above 0"
    assert 0 <= alive <= start, "live fish cannot outnumber the start"
    return alive / start * 100


print(f"{survival_percent(1000, 940):.1f}%")
```

```
94.0%
```

```
# Chapter 12 - A failing assert: our assumption turned out to be wrong
def survival_percent(start, alive):
    assert start > 0, "the starting count must be above 0"
    assert 0 <= alive <= start, "live fish cannot outnumber the start"
    return alive / start * 100


print(f"{survival_percent(1000, 1040):.1f}%")
```

```
Traceback (most recent call last):
  File "09_assert_fails.py", line 8, in <module>
    print(f"{survival_percent(1000, 1040):.1f}%")
             ~~~~~~~~~~~~~~~~~^^^^^^^^^^^^
  File "09_assert_fails.py", line 4, in survival_percent
    assert 0 <= alive <= start, "live fish cannot outnumber the start"
           ^^^^^^^^^^^^^^^^^^^
AssertionError: live fish cannot outnumber the start
```

Live fish (1040) outnumber the starting fish (1000), which makes no sense, so the second `assert` stops the program with the message we wrote. Use `assert` to catch **programmer mistakes**, not mistakes in user input.

> ⚠ WARNING
>
> Python can be run with the `-O` option, which **removes every `assert`**. We tried it on the script above: with `python -O`, the result `104.0%` is printed with no error. So never use `assert` as the only checker of user input; use `if` and `raise`.

3. logging: a tidier print, with importance **levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL) that can be filtered.

```
# Chapter 12 - logging: a work diary of the program with levels of importance
import logging
import sys

logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
                    format="%(levelname)-8s %(message)s")

logging.debug("checking 3 data rows")
logging.info("pond A data was read")
logging.warning("pond B pH = 9.8, too high")
logging.error("pond C data was not found")

logging.getLogger().setLevel(logging.WARNING)       # now only WARNING and above
logging.info("this message does not appear")
logging.warning("this message appears")
```

```
DEBUG    checking 3 data rows
INFO     pond A data was read
WARNING  pond B pH = 9.8, too high
ERROR    pond C data was not found
WARNING  this message appears
```

After the level is raised to WARNING, the `info` message no longer appears, but `warning` still does. With `logging`, you don't need to delete debug messages when the program is finished; just raise the level. (More about logging is in Chapter 19.)

The fourth tool is Python's built-in **debugger** `pdb`: the `breakpoint()` function stops the program at one line and opens a `(Pdb)` prompt where you can type `p variable` (see its content), `n` (next line), `c` (continue), and `q` (quit).

```
# Chapter 12 - The interactive debugger pdb: stop at one line and peek inside variables
def average(numbers):
    total = sum(numbers)
    breakpoint()                  # the program stops here and opens a (Pdb) prompt
    return total / len(numbers)


print(average([32.5, 45.2, 28.0]))
```

This code was **not run** when the book was built because pdb waits for the user to type, so no output is shown. Try it yourself in a terminal: run the file, then type `p total` and `c`.

Running Project B3: resistant to wrong input

Two project scripts. First, **ask for input until it is right**. The pretend input (supplied by a `.input` file) holds five answers: `abc`, `-5`, an empty line, `450`, and finally `32,5`.

```
# Chapter 12 - Project B3: ask for a number until it is right (resistant to wrong input)
def ask_diameter():
    while True:
        text = input("Tree diameter (cm): ")
        try:
            value = float(text.replace(",", "."))      # accept "32,5" and "32.5"
        except ValueError:
            print("  Not a number, try again.")
            continue
        if not 1 <= value <= 300:
            print("  Must be between 1 and 300 cm, try again.")
            continue
        return value
```

```
result = ask_diameter()
print("Diameter recorded:", result)
```

```
Tree diameter (cm): abc
  Not a number, try again.
Tree diameter (cm): -5
  Must be between 1 and 300 cm, try again.
Tree diameter (cm):
  Not a number, try again.
Tree diameter (cm): 450
  Must be between 1 and 300 cm, try again.
Tree diameter (cm): 32,5
Diameter recorded: 32.5
```

The first four answers are rejected for different reasons (not a number, out of range), and `32,5` is accepted after the comma is replaced with a dot. The pattern `while True` + `try/except` + `continue` + `return` is the standard recipe for "repeat until right".

Second, **cleaning a jumble of data rows** that will later come from files (Chapters 13 and 14). There are six rows; three of them are broken:

```
# Chapter 12 - Project B3: cleaning a jumble of data rows without collapsing
import logging
import sys

logging.basicConfig(stream=sys.stdout, level=logging.WARNING, format="%(levelname)s:
%(message)s")


class BadRow(Exception):
    pass


def read_row(text):
    """Turns 'plot,species,diameter' into a dict, or raises BadRow."""
    parts = [p.strip() for p in text.split(",")]
    if len(parts) != 3:
        raise BadRow(f"needs 3 columns, got {len(parts)}: {text!r}")
    plot, species, diameter = parts
    try:
        diameter = float(diameter.replace(",", "."))
    except ValueError:
        raise BadRow(f"diameter is not a number: {diameter!r}") from None
    if not 1 <= diameter <= 300:
        raise BadRow(f"diameter outside 1-300: {diameter}")
    return {"plot": plot, "species": species, "diameter_cm": diameter}


raw = ["A,Teak,32.5", "A,Meranti,45.2", "B,Mahogany,twenty", "B,Sengon", "C,Teak,-3",
"C,Teak,36.0"]
clean, rejected = [], 0
for row in raw:
    try:
        clean.append(read_row(row))
    except BadRow as e:
        logging.warning("row rejected: %s", e)
        rejected += 1

print(f"Accepted {len(clean)} rows, rejected {rejected}")
```

```
for c in clean:
    print(c)
```

```
WARNING: row rejected: diameter is not a number: 'twenty'
WARNING: row rejected: needs 3 columns, got 2: 'B,Sengon'
WARNING: row rejected: diameter outside 1-300: -3.0
Accepted 3 rows, rejected 3
{'plot': 'A', 'species': 'Teak', 'diameter_cm': 32.5}
{'plot': 'A', 'species': 'Meranti', 'diameter_cm': 45.2}
{'plot': 'C', 'species': 'Teak', 'diameter_cm': 36.0}
```

The flow: `read_row` raises `BadRow` for each kind of damage; the loop catches it, **logs a warning** with `logging`, and moves on to the next row. End result: 3 rows accepted, 3 rejected, and the program never collapses. The `from None` part hides the original error (`ValueError`) so our own message is the one shown; without it, the traceback would hold two errors that confuse the reader.

Exercises

1. Write a function `to_number(text)` that returns a `float` from the text, or `None` if the text is not a number. Test it with `"12,5"`, `"abc"`, and `""`.
2. Change `13_project_error_proof.py` so that a diameter above 150 cm is not rejected, but only produces `logging.warning("very large tree")`.
3. Add an `assert` to a function `average(numbers)` that makes sure the list is not empty. Compare its error message with the `ZeroDivisionError` in the traceback earlier.
4. **Challenge:** make an error class `UnreasonablePh(FieldDataError)` and a function `check_ponds(items)` that collects **all** problem ponds (instead of stopping at the first), then prints a summary.

Chapter Summary

- **`try`** / **`except`** catches errors; **`else`** runs when there is no error; **`finally`** always runs.
- Catch specific error types; avoid an empty `except:`.
- **`raise`** raises your own error; a new error type is made by inheriting from `Exception`.
- A **traceback** is read from the bottom: type and message, scene of the incident, then the chain of calls.
- Debugging tools: `print` (with `repr`), `assert` (can be switched off with `-O`), `logging`, and `pdb`/`breakpoint()`.
- Project B3: a program that asks for input until it is right and rejects broken data rows without collapsing.

Next, in Chapter 13: files and basic I/O, how a program saves and reads data so it isn't lost when closed.

Chapter 13 — Files and Basic I/O: Data That Survives Closing the Program

Series B3 — Small Programs That Are Useful

In this chapter, you will:

- Write and read text files with `open()` and `with`
- Choose the right mode: `"r"`, `"w"`, `"a"`, `"x"`
- Manage files and folders with `pathlib`
- Understand **encoding** (how text is stored as bytes) and why it should be UTF-8
- Face a file that does not exist, and save tree data to a file and read it back

All scripts for this chapter are in the `kode/ch13/` folder. On the web version, you can also download them all at once (zip). Scripts that create files write them to a `_tmp/` subfolder next to the script, so your folder stays tidy.

So far, all data has lived in the computer's **memory**, like notes on a whiteboard: once the program is closed, the writing is wiped. A good field officer writes in a **notebook**, which is a **file** on the disk, still there when the computer is switched off. This chapter teaches your program to write and read that notebook. The common term is **I/O** (*input/output*): data comes in from a file, data goes out to a file.

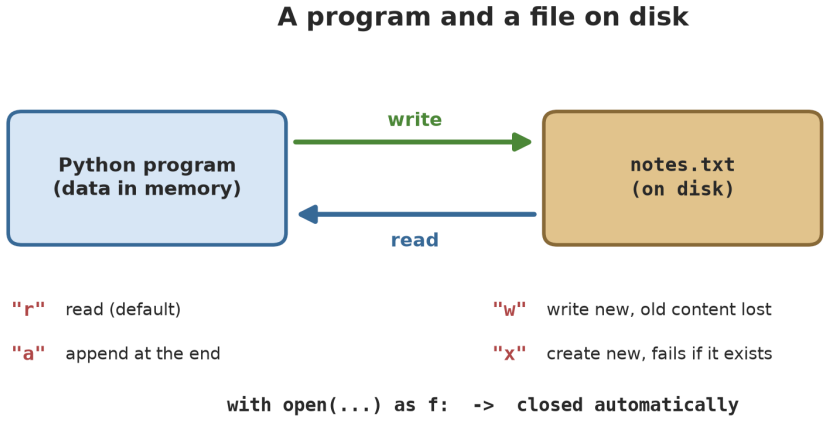


Figure 0.1: A program writes to a file on disk (write) and reads it back later (read). Modes: r = read, w = write new, a = append, x = create new only.

Writing a text file

```
# Chapter 13 - Writing a text file with open() and with
from pathlib import Path

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

with open(file, "w", encoding="utf-8") as f:            # "w" = write (old content
is erased)
```

```
    f.write("Field notes 1 October\n")
    f.write("Plot A: 12 teak trees\n")
    f.write("Plot B: 9 meranti trees\n")

print("File written:", file.name)
print("Size:", file.stat().st_size, "bytes")
```

```
File written: notes.txt
Size: 71 bytes
```

- `open(file, "w", encoding="utf-8")` opens a file for **writing** (`"w"`). If the file already exists, its old content is **erased**.
- `with ... as f:` opens the file and **closes it automatically** when the block ends. You will see why that matters below.
- `f.write(...)` writes text. It does **not** add a new line by itself, so you write `\n` at the end of each line.

Why is the size 71 bytes when the letters count only 68? On Windows, every `\n` is written as two bytes (`\r\n`), and there are three lines, so 68 + 3 = 71. On macOS or Linux the result is 68. Don't be surprised if file sizes differ between computers.

Reading a text file

```
# Chapter 13 - Four ways to read a text file
from pathlib import Path

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
file = tmp / "notes.txt"
file.write_text("Field notes 1 October\nPlot A: 12 teak trees\nPlot B: 9 meranti
trees\n",
                encoding="utf-8")

with open(file, "r", encoding="utf-8") as f:
    everything = f.read()                  # 1. the whole content as one text
print(repr(everything))

with open(file, encoding="utf-8") as f:
    print(repr(f.readline()))              # 2. one line (with the \n at the end)

with open(file, encoding="utf-8") as f:
    lines = f.readlines()                  # 3. all lines as a list
print(len(lines), "lines")

with open(file, encoding="utf-8") as f:
    for number, line in enumerate(f, start=1):      # 4. line by line (saves memory)
        print(number, line.rstrip())
```

```
'Field notes 1 October\nPlot A: 12 teak trees\nPlot B: 9 meranti trees\n'
'Field notes 1 October\n'
3 lines
1 Field notes 1 October
2 Plot A: 12 teak trees
3 Plot B: 9 meranti trees
```

Four ways to read, from the bluntest to the most economical:

1. `read()`: the whole content as **one text**. Note the `\n` between lines (`repr` makes them visible).
2. `readline()`: **one line**, with the `\n` at the end. Call it again and it gives the next line.
3. `readlines()`: **all lines as a list**.

4. **A `for` loop directly over the file**: one line per round, without loading everything into memory. This is the best choice for big files. `rstrip()` removes the `\n` at the end, and `enumerate(f, start=1)` supplies line numbers.

Modes: w, a, x

```
# Chapter 13 - Mode "w" overwrites, mode "a" appends, mode "x" refuses to overwrite
from pathlib import Path

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

with open(file, "w", encoding="utf-8") as f:
    f.write("Harvest 1: 245.5 kg\n")
with open(file, "a", encoding="utf-8") as f:          # add at the end
    f.write("Harvest 2: 310.0 kg\n")
print(file.read_text(encoding="utf-8"))

with open(file, "w", encoding="utf-8") as f:         # overwrites: old content is gone
    f.write("Harvest 3: 198.5 kg\n")
print(file.read_text(encoding="utf-8"))

try:
    with open(file, "x", encoding="utf-8") as f:     # "x": fails if it already exists
        f.write("never mind")
except FileExistsError:
    print("Mode x refuses: the file already exists")
```

```
Harvest 1: 245.5 kg
Harvest 2: 310.0 kg

Harvest 3: 198.5 kg

Mode x refuses: the file already exists
```

Mode	Meaning
`"r"`	read (the default); error if the file does not exist
`"w"`	write; **overwrites** the old content
`"a"`	append at the end; old content is safe
`"x"`	create new; **fails** if the file already exists

See the result: after mode `"a"`, there are two harvests. After mode `"w"`, harvests 1 and 2 are **gone**, leaving only harvest 3. Mode `"x"` protects important files from being overwritten by accident (it raises `FileExistsError`, which we catch with the `try` of Chapter 12).

> ⚠ WARNING
>
> Mode `"w"` erases old content **without asking**. If you want to add to a daily log, use `"a"`. Choosing the wrong letter can wipe out days of data.

Why with

```
# Chapter 13 - Why with: the file is closed automatically, even when an error happens
from pathlib import Path
```

```
tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
file = tmp / "example.txt"

f = open(file, "w", encoding="utf-8")       # the old way: you must close it yourself
f.write("hello\n")
print("Before close, closed?", f.closed)
f.close()
print("After close, closed?", f.closed)

try:
    with open(file, "w", encoding="utf-8") as g:
        g.write("hello again\n")
        raise RuntimeError("something went wrong in the middle of writing")
except RuntimeError as e:
    print("Error caught:", e)
print("File still closed neatly?", g.closed)
```

```
Before close, closed? False
After close, closed? True
Error caught: something went wrong in the middle of writing
File still closed neatly? True
```

An opened file must be **closed** so its content is really saved and it doesn't lock the file for other programs. The old way: `open()` then `f.close()` yourself; if an error happens in between, `close()` may be skipped. With `with`, closing is guaranteed: in the example, a `RuntimeError` blows up in the middle of writing, yet `g.closed` is still `True`.

> **i** REMEMBER
>
> Make `with open(...)` your **constant** habit. You will almost never need to write `close()` yourself.

pathlib for files and folders

In Chapter 11 you met `Path`. Now it works on a real disk:

```
# Chapter 13 - pathlib for real files and folders
from pathlib import Path

tmp = Path(__file__).parent / "_tmp" / "field"
tmp.mkdir(parents=True, exist_ok=True)       # make nested folders; no error if it
already exists

for name in ["plot_a.txt", "plot_b.txt", "fish.csv"]:
    (tmp / name).write_text(f"content of {name}\n", encoding="utf-8")

print("Folder exists?", tmp.is_dir())
print("All files    :", sorted(p.name for p in tmp.iterdir()))
print("Only .txt    :", sorted(p.name for p in tmp.glob("*.txt")))
print("Size of plot_a:", (tmp / "plot_a.txt").stat().st_size, "bytes")

old = tmp / "fish.csv"
new = old.rename(tmp / "fish_2026.csv")
print("Renamed to   :", new.name)
new.unlink()                                 # delete one file (in our own _tmp folder)
print("Remaining    :", sorted(p.name for p in tmp.iterdir()))
```

```
Folder exists? True
All files    : ['fish.csv', 'plot_a.txt', 'plot_b.txt']
Only .txt    : ['plot_a.txt', 'plot_b.txt']
Size of plot_a: 23 bytes
Renamed to   : fish_2026.csv
Remaining    : ['plot_a.txt', 'plot_b.txt']
```

- `mkdir(parents=True, exist_ok=True)` makes nested folders and doesn't complain if they exist.
- `write_text()` and `read_text()` are shortcuts for writing or reading the whole content of a small file, without `with`.
- `iterdir()` lists a folder's content; `glob("*.txt")` picks files matching a pattern. We arrange them with `sorted` because the order from the disk is not guaranteed.
- `rename()` changes the name, `unlink()` deletes one file, `stat().st_size` gives the size in bytes.

Use `unlink()` carefully: a file deleted this way does **not go to the recycle bin**. In the example, we only delete a file we made ourselves in the `_tmp` folder.

Encoding: when text turns into strange symbols

A disk stores only **bytes** (numbers 0 to 255), not letters. **Encoding** is the rule that translates letters into bytes and back. The most recommended rule is **UTF-8**, which can write any letter, including `°`, `±`, and non-Latin scripts.

```
# Chapter 13 - Text is stored as bytes; use UTF-8 and write it out explicitly
from pathlib import Path

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

text = "Water temp 28 °C, Pontianak, result ± 5%"
file.write_text(text, encoding="utf-8")

raw = file.read_bytes()                      # the file content as-is: a row of bytes
print("Number of characters:", len(text))
print("Number of bytes     :", len(raw))
print("Bytes of degree sign:", "°".encode("utf-8"))
print("Read back           :", file.read_text(encoding="utf-8"))
```

```
Number of characters: 40
Number of bytes     : 42
Bytes of degree sign: b'\xc2\xb0'
Read back           : Water temp 28 °C, Pontianak, result ± 5%
```

That text is 37 characters but 39 bytes: the `°` and `±` symbols each take **two bytes** in UTF-8 (`b'\xc2\xb0'` for `°`). That is why "number of characters" and "number of bytes" can differ.

Now what happens when a file written with one rule is read with another:

```
# Chapter 13 - The WRONG text rule: written with one rule, read with another
from pathlib import Path

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
file = tmp / "temperature_utf8.txt"
file.write_text("Water temp 28 °C, result ± 5%", encoding="utf-8")

# cp1252 = the default rule on many Western-language Windows computers
print("Read as cp1252:", file.read_text(encoding="cp1252"))

try:
```

```
    file.read_text(encoding="ascii")          # ascii knows only 128 symbols
except UnicodeDecodeError as e:
    print("Read as ascii -> UnicodeDecodeError:", e.reason)

print("Read as utf-8 :", file.read_text(encoding="utf-8"))
```

```
Read as cp1252: Water temp 28 Â°C, result Â± 5%
Read as ascii -> UnicodeDecodeError: ordinal not in range(128)
Read as utf-8 : Water temp 28 °C, result ± 5%
```

The results `Â°C` and `Â±` are **mojibake**: two UTF-8 bytes read as two separate letters by the `cp1252` rule. With the `ascii` rule, Python refuses outright with a `UnicodeDecodeError`. Only reading as UTF-8 returns the original text.

> ⚠ WARNING
>
> If you **don't state** `encoding=`, Python uses the system default. On this book author's computer (Windows), the default is `cp1252`; we checked it directly with `locale.getencoding()`. As a result, a file written on one computer may be garbled when read on another. Starting with Python 3.15 (released 9 October 2026), UTF-8 is the default: we proved it by opening a file without `encoding=` in Python 3.15.0, which gave `utf-8`, and switching it off with `PYTHONUTF8=0` brought back `cp1252` (see Chapter 47). This book uses Python 3.14, which still follows the system rule, so **always write `encoding="utf-8"`**, so your code is right on every version and computer.

A file that does not exist

```
# Chapter 13 - Pitfall: opening a file that does not exist
with open("tree_data_missing.txt", encoding="utf-8") as f:
    print(f.read())
```

```
Traceback (most recent call last):
  File "08_file_not_found.py", line 2, in <module>
    with open("tree_data_missing.txt", encoding="utf-8") as f:
         ~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'tree_data_missing.txt'
```

`FileNotFoundError` is the most common error in file work: a typo in the name, the wrong folder, or the file simply hasn't been made. Two polite ways to deal with it:

```
# Chapter 13 - Facing a missing file politely
from pathlib import Path


def read_or_fallback(file, fallback="(no data yet)"):
    try:
        return Path(file).read_text(encoding="utf-8")
    except FileNotFoundError:
        print(f"  [info] {Path(file).name} not found, using the fallback")
        return fallback


print(read_or_fallback("tree_data_missing.txt"))

# Second way: check first with exists()
address = Path("tree_data_missing.txt")
if address.exists():
    print(address.read_text(encoding="utf-8"))
```

```
else:
    print("exists() check: the file is not there yet, skip")
```

```
  [info] tree_data_missing.txt not found, using the fallback
(no data yet)
exists() check: the file is not there yet, skip
```

The first way uses `try/except` (recommended, because a file can vanish between being "checked" and being "opened"). The second uses `exists()`: easier to read, but it only checks the state at that moment.

Running Project B3: saving and loading tree data

Now we combine Chapters 12 and 13. The script saves six trees to `trees.txt` (one tree per line, columns separated by `|`) and **deliberately slips in one broken line**. Then the function `load_trees` reads it back, resistant to mistakes:

```
# Chapter 13 - Project B3: tree data saved to a text file and read back
from pathlib import Path

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

trees = [
    ("A", "Teak", 32.5), ("A", "Meranti", 45.2), ("B", "Mahogany", 28.0),
    ("B", "Sengon", 51.5), ("C", "Teak", 36.0), ("C", "Meranti", 40.8),
]

# 1. Save: one tree per line, columns separated by the | sign
with open(file, "w", encoding="utf-8") as f:
    for plot, species, diameter in trees:
        f.write(f"{plot}|{species}|{diameter}\n")
    f.write("C|Teak|broken\n")                # one broken line, inserted on purpose


# 2. Read it back, resistant to mistakes (the lesson of Chapter 12)
def load_trees(address):
    result, rejected = [], 0
    try:
        with open(address, encoding="utf-8") as f:
            for line in f:
                try:
                    plot, species, diameter = line.strip().split("|")
                    result.append((plot, species, float(diameter)))
                except ValueError:
                    rejected += 1
    except FileNotFoundError:
        print("File does not exist yet, starting with empty data")
    return result, rejected


data, rejected = load_trees(file)
print(f"Read {len(data)} trees, {rejected} line rejected")
print("Mean diameter:", round(sum(d for _, _, d in data) / len(data), 2), "cm")
print("First record:", data[0])
print("A file that does not exist:", load_trees(tmp / "missing.txt"))
```

```
Read 6 trees, 1 line rejected
Mean diameter: 39.0 cm
```

```
First record: ('A', 'Teak', 32.5)
File does not exist yet, starting with empty data
A file that does not exist: ([], 0)
```

Six trees are read, the line `C|Teak|broken` is rejected (its diameter is not a number), and the mean diameter is 39.0 cm (matching the result of Chapter 11). A file that doesn't exist gives empty data with a friendly message, not an error. But this home-made `|` format is fragile: what if a species name contains a `|`? In Chapter 14 we use standard formats (CSV and JSON) that have already solved that problem.

Exercises

1. Write a script that writes the numbers 1 to 10 (one per line) to `numbers.txt`, then reads it back and prints the sum.
2. Run the script from exercise 1 twice with mode `"a"` (not `"w"`). How many lines does the file hold, and why?
3. Change `07_wrong_encoding.py`: write with `encoding="utf-16"`, then read with `"utf-8"`. What happens? Catch the error with `try`.
4. **Challenge:** write a function `log_harvest(file, date, kg)` that appends one line `date|kg` to a file (mode `"a"`), and a function `total_harvest(file)` that adds up the kg column while skipping broken lines.

Chapter Summary

- A **file** keeps data on disk; a program reads it (`"r"`) and writes it (`"w"`, `"a"`, `"x"`).
- **Always** `with open(..., encoding="utf-8") as f:`: the file closes automatically.
- Reading: `read()`, `readline()`, `readlines()`, and a `for` loop over the file (memory-friendly).
- `pathlib.Path`: `mkdir`, `read_text`, `write_text`, `iterdir`, `glob`, `rename`, `unlink`, `exists`.
- A wrong **encoding** gives strange symbols or `UnicodeDecodeError`; always write `encoding="utf-8"`.
- A file that doesn't exist: catch `FileNotFoundError`.

Next, in Chapter 14: standard data formats (CSV and JSON) so our data is tidy and readable by other programs.

Chapter 14 — Data Formats: CSV, JSON, and Be Careful with Pickle

Series B3 — Small Programs That Are Useful

In this chapter, you will:

- Save and read tables with the `csv` module
- Save and read nested data with the `json` module
- Understand **serialisation**, and know why values from CSV are always text
- Understand the danger of `pickle` and prove it safely
- Save the data of all five sectors to CSV and JSON, then load it back resistant to mistakes

All scripts for this chapter are in the `kode/ch14/` folder. On the web version, you can also download them all at once (zip). Result files are written to the `_tmp/` subfolder.

In Chapter 13, we saved tree data in a home-made format (`A|Teak|32.5`) and already suspected the format was fragile. Imagine two officers sending each other notes: if each has their own way of writing, neither can read the other's notes. So the world agrees on a few **standard formats**. Turning data in memory into text (or bytes) that can be stored is called **serialisation**; turning it back is **deserialisation**. This chapter covers three formats: CSV, JSON, and pickle.

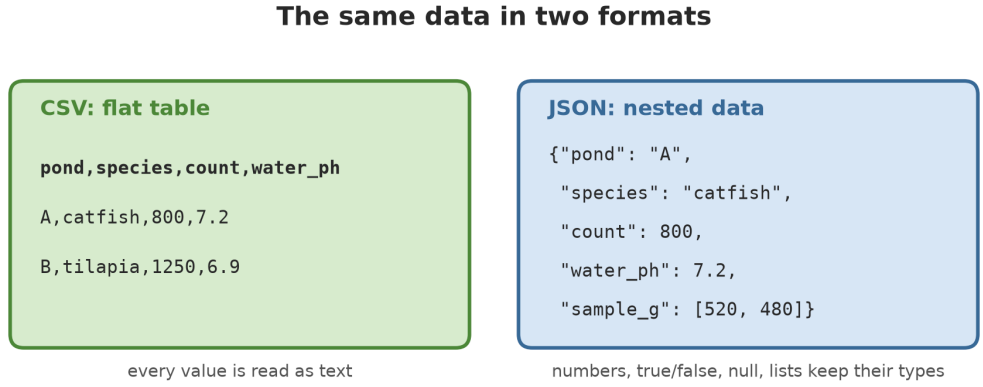


Figure 0.1: CSV stores data as a flat table; JSON stores nested data with brackets. Both are plain text.

CSV: a table as text

CSV (*comma-separated values*) is a table where each row is one record and the columns are separated by commas. Excel can open the file. We write the pond data from Chapter 11:

```
# Chapter 14 - Writing CSV: a table as comma-separated text
import csv
from pathlib import Path

from core_data import fish

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
file = tmp / "fish.csv"
```

```
columns = ["pond", "species", "count", "avg_weight_g", "water_ph"]
with open(file, "w", newline="", encoding="utf-8") as f:      # newline="" is required
for csv
    writer = csv.DictWriter(f, fieldnames=columns)
    writer.writeheader()                                       # the column title row
    writer.writerows(fish)                                     # all rows at once

print(file.read_text(encoding="utf-8"))
```

```
pond,species,count,avg_weight_g,water_ph
A,catfish,800,540.5,7.2
B,tilapia,1250,300.0,6.9
C,catfish,600,480.0,7.0
```

- `DictWriter` writes a list of dicts; `fieldnames` sets the **column order**.
- `writeheader()` writes the title row, `writerows()` writes all rows.
- `newline=""` in `open()` is **required** for the `csv` module; the Python documentation says a CSV file should be opened with `newline=''` so line endings are handled correctly.

This is fictional data (sample numbers).

Reading CSV: everything is text

```
# Chapter 14 - Reading CSV: every value arrives as TEXT
import csv
from pathlib import Path

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
file = tmp / "fish_read.csv"
file.write_text("pond,species,count,avg_weight_g\nA,catfish,800,540.5\nB,tilapia,1250,300.0\n",
                encoding="utf-8")

with open(file, newline="", encoding="utf-8") as f:
    rows = list(csv.DictReader(f))            # each row becomes a dict keyed by the
column titles

print(rows[0])
print("Type of count:", type(rows[0]["count"]).__name__)
print("Wrong  :", rows[0]["count"] + rows[1]["count"])       # texts joined, not added

for r in rows:                                # convert the types ourselves
    r["count"] = int(r["count"])
    r["avg_weight_g"] = float(r["avg_weight_g"])
print("Right  :", rows[0]["count"] + rows[1]["count"])
```

```
{'pond': 'A', 'species': 'catfish', 'count': '800', 'avg_weight_g': '540.5'}
Type of count: str
Wrong  : 8001250
Right  : 2050
```

`DictReader` makes a dict per row with keys taken from the title row. Notice: `'count': '800'` is wrapped in quotes, meaning **text**, not a number. The `csv` documentation states that no automatic type conversion happens. As a result `"800" + "1250"` gives `8001250` (text joined), not 2050. **You must convert the types yourself** with `int()` and `float()`.

> ⚠ WARNING
>
> The values `True` and `False` in CSV are text too. `bool("False")` gives `True` (because non-empty text counts as true)! Match the text explicitly, as in the project below.

Why not `split(",")`

```
# Chapter 14 - A comma inside the content: why we use the csv module, not split(",")
import csv
import io

raw = 'plot,note\nA,"near the river, peat soil"\nB,"locals call it the ""sacred
forest"""\n'
print("Raw content:")
print(raw)

print("The naive split:")
for line in raw.splitlines()[1:]:
    print("  ", line.split(","))

print("Using csv.reader:")
for row in csv.reader(io.StringIO(raw)):
    print("  ", row)
```

```
Raw content:
plot,note
A,"near the river, peat soil"
B,"locals call it the ""sacred forest"""

The naive split:
   ['A', '"near the river', ' peat soil"']
   ['B', '"locals call it the ""sacred forest"""']
Using csv.reader:
   ['plot', 'note']
   ['A', 'near the river, peat soil']
   ['B', 'locals call it the "sacred forest"']
```

The note “near the river, peat soil” contains a comma. CSV wraps such content in quote marks, and quote marks inside the content are doubled (`""`). `split(",")` is blind to that rule and cuts the note into three wrong pieces; `csv.reader` handles it correctly. This is the lesson of Chapter 13: a home-made format will collide with cases you didn’t expect, while a standard module has already thought of them.

JSON: nested data

CSV only suits flat tables. For nested data (for example a dict holding a list), use **JSON** (*JavaScript Object Notation*), a text format used by almost every internet service.

```
# Chapter 14 - JSON: nested data as text
import json

pond = {"pond": "A", "species": "catfish", "count": 800, "water_ph": 7.2,
        "active": True, "note": None, "sample_g": [520, 480, 560]}

text = json.dumps(pond)                      # dict -> JSON text
print(text)
print(json.dumps(pond, indent=2, ensure_ascii=False)[:90], "...")
```

```
back = json.loads(text)                      # JSON text -> dict
print(back == pond, type(back).__name__)

print(json.dumps({"temp": "28 °C"}))                       # default: special letters
are escaped
print(json.dumps({"temp": "28 °C"}, ensure_ascii=False))      # shown as they are
print(json.dumps((1, 2)), json.dumps({1: "a"}))               # tuple becomes list,
number key becomes text
```

```
{"pond": "A", "species": "catfish", "count": 800, "water_ph": 7.2, "active": true,
"note": null, "sample_g": [520, 480, 560]}
{
  "pond": "A",
  "species": "catfish",
  "count": 800,
  "water_ph": 7.2,
  "active": tr ...
True dict
{"temp": "28 \u00b0C"}
{"temp": "28 °C"}
[1, 2] {"1": "a"}
```

- `json.dumps()` turns a Python object into JSON **text** (`s` = *string*); `json.loads()` does the reverse.
- Type equivalents: `True` becomes `true`, `None` becomes `null`, a dict becomes an object `{...}`, a list becomes an array `[...]`. JSON text uses **double quote marks**.
- `indent=2` tidies the layout so humans can read it easily.
- `ensure_ascii=False` shows `°` as it is; without it, JSON writes `°` (still valid, just less pleasant to read).
- Two surprises on the last line: the tuple `(1, 2)` comes out as a **list**, and the number key `1` becomes the text `"1"`. JSON has no tuples, and its keys are always text.

JSON to a file

```
# Chapter 14 - JSON to and from a file, with nested data of five sectors
import json
from pathlib import Path

from core_data import trees, crops, fish, livestock

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

everything = {"trees": trees, "crops": crops, "fish": fish, "livestock": livestock}

with open(file, "w", encoding="utf-8") as f:
    json.dump(everything, f, indent=2, ensure_ascii=False)

with open(file, encoding="utf-8") as f:
    back = json.load(f)

print("Exactly the same?", back == everything)
print("Parts:", list(back))
print("First tree:", back["trees"][0])
print("Type of healthy:", type(back["trees"][2]["healthy"]).__name__)
print("File size:", file.stat().st_size, "bytes")
```

```
Exactly the same? True
Parts: ['trees', 'crops', 'fish', 'livestock']
```

```
First tree: {'species': 'Teak', 'diameter_cm': 32.5, 'height_m': 18.0, 'healthy': True,
'plot': 'A'}
Type of healthy: bool
File size: 2222 bytes
```

Unlike CSV, JSON **gives the types back**: `healthy` returns as a `bool`, numbers return as `int` or `float`, and the whole structure (four tables in one dict) returns **exactly the same** as the original. (The file size was measured on Windows and may differ slightly on another computer because of line endings.) For saving one complete set of data, JSON is usually more comfortable; for a big table that will be opened in Excel, CSV suits better.

Two JSON problems

```
# Chapter 14 - Two common JSON problems: broken text and an unknown type
import json
from datetime import date

broken = '{"pond": "A", "count": 800,}'       # extra comma at the end: JSON does not
allow it
try:
    json.loads(broken)
except json.JSONDecodeError as e:
    print("Broken JSON:", type(e).__name__, "on line", e.lineno)

record = {"pond": "A", "stocking_date": date(2026, 3, 1)}
try:
    json.dumps(record)
except TypeError as e:
    print("Cannot:", e)

print("With default=str:", json.dumps(record, default=str))
print("Better habit: store the date as ISO text ->", date(2026, 3, 1).isoformat())
```

```
Broken JSON: JSONDecodeError on line 1
Cannot: Object of type date is not JSON serializable
With default=str: {"pond": "A", "stocking_date": "2026-03-01"}
Better habit: store the date as ISO text -> 2026-03-01
```

1. **Broken text**: an extra comma at the end causes a `JSONDecodeError`. This often happens when JSON is written or edited by hand. The full message names the line and column; we only show the error name and line number because the wording of the message differs between Python versions.
2. **Unknown type**: JSON doesn't know dates, so `json.dumps` raises a `TypeError`. A quick way out: `default=str`. A better habit: store dates as ISO text (`2026-03-01`) and convert back with `date.fromisoformat()` (Chapter 11).

Pickle: handy, but dangerous

Pickle stores almost **anything** from Python (your own objects, functions, sets) as bytes. Handy, but the Python documentation states plainly that the `pickle` module is **not secure** and should only be used on data you trust, because maliciously designed pickle data can **run any code** when opened. Let's prove it, **in a safe way**: we make a pickle that only runs `print`, with no harmful command.

```
# Chapter 14 - pickle: handy for your own data, DANGEROUS for data from strangers
import pickle
import pickletools


class Carrier:
```

```
    """An object that deliberately makes the pickle RUN a function when opened."""
    def __reduce__(self):
       # In a real attack this could be a destructive command. Here it is just a print.
        return (print, ("!!! this code ran only because pickle.loads() was called",))


safe = {"pond": "A", "count": 800}
safe_package = pickle.dumps(safe)
print("Own-data pickle opened:", pickle.loads(safe_package))

risky_package = pickle.dumps(Carrier(), protocol=4)    # protocol fixed so the output
is the same on every version
print("Before loads: nothing has run yet")
pickle.loads(risky_package)                    # <- this is the danger
print("After loads: the print above was run by pickle, not by us")

print("--- looking inside the pickle WITHOUT opening it (pickletools) ---")
pickletools.dis(risky_package)
```

```
Own-data pickle opened: {'pond': 'A', 'count': 800}
Before loads: nothing has run yet
!!! this code ran only because pickle.loads() was called
After loads: the print above was run by pickle, not by us
--- looking inside the pickle WITHOUT opening it (pickletools) ---
    0: \x80 PROTO      4
    2: \x95 FRAME      85
   11: \x8c SHORT_BINUNICODE 'builtins'
   21: \x94 MEMOIZE    (as 0)
   22: \x8c SHORT_BINUNICODE 'print'
   29: \x94 MEMOIZE    (as 1)
   30: \x93 STACK_GLOBAL
   31: \x94 MEMOIZE    (as 2)
   32: \x8c SHORT_BINUNICODE '!!! this code ran only because pickle.loads() was called'
   90: \x94 MEMOIZE    (as 3)
   91: \x85 TUPLE1
   92: \x94 MEMOIZE    (as 4)
   93: R    REDUCE
   94: \x94 MEMOIZE    (as 5)
   95: .    STOP
highest protocol among opcodes = 4
```

Follow the result:

1. A pickle of our own data (the pond dict) opens normally.
2. We make a pickle from a `Carrier` object whose `__reduce__` says "please call this function when the pickle is opened".
3. Between our two `print` lines, the line `!!! this code ran ...` appears. **We did not call that print; `pickle.loads()` did.** If it had been a command to delete files, the result would be the same: it runs without permission.
4. `pickletools.dis()` reads the pickle **without running it** and shows clearly that it calls `builtins.print` (`STACK_GLOBAL` then `REDUCE`). The Python documentation also recommends `python -m pickletools` as a safer way than opening a pickle to inspect it.

> ⚠ WARNING
>
> **Never** open (`pickle.load` or `pickle.loads`) a pickle file that comes from other people, from the internet, or from email, and never accept one from users of a web application. For outside data, use **JSON** or CSV. Pickle is only for data that **you yourself** wrote and keep in a safe place.

Running Project B3: save and load five sectors

The data of the five sectors (forest, orchard, field crops, fisheries, livestock; orchard and field crops share the `crops` table) is now saved to CSV and JSON. The tools are in the module `data_tools.py`: `save_csv`, `load_csv` (with a **schema** that names the type converter of each column), `save_json`, and `load_json`.

```
# Chapter 14 - Project B3: five sectors saved to CSV and JSON, read back, compared
from pathlib import Path

from data_tools import save_csv, load_csv, save_json, load_json
from core_data import trees, crops, fish, livestock

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

tables = {"trees": trees, "crops": crops, "fish": fish, "livestock": livestock}
bool_text = lambda s: {"True": True, "False": False}[s]         # KeyError if not
True/False
schema = {
     "trees": {"species": str, "diameter_cm": float, "height_m": float, "healthy":
bool_text, "plot": str},
    "crops": {"crop": str, "block": str, "age_days": int, "yield_kg": float},
      "fish": {"pond": str, "species": str, "count": int, "avg_weight_g": float,
"water_ph": float},
    "livestock": {"pen": str, "species": str, "age_months": int, "weight_kg": float},
}

for name, rows in tables.items():
    save_csv(tmp / f"{name}.csv", rows)
save_json(tmp / "inventory.json", tables)

for name in tables:
    loaded, rejected = load_csv(tmp / f"{name}.csv", schema[name])
    same = loaded == tables[name]
    print(f"{name:10} CSV: {len(loaded)} rows, rejected {rejected}, same as original:
{same}")

print("Whole JSON same as original:", load_json(tmp / "inventory.json") == tables)
print("Content of trees.csv (first 3 lines):")
for line in (tmp / "trees.csv").read_text(encoding="utf-8").splitlines()[:3]:
    print(line)
```

```
trees      CSV: 6 rows, rejected 0, same as original: True
crops      CSV: 4 rows, rejected 0, same as original: True
fish       CSV: 3 rows, rejected 0, same as original: True
livestock  CSV: 4 rows, rejected 0, same as original: True
Whole JSON same as original: True
Content of trees.csv (first 3 lines):
species,diameter_cm,height_m,healthy,plot
Teak,32.5,18.0,True,A
Meranti,45.2,27.5,True,A
```

The result `same as original: True` for all four tables proves that data coming back from CSV (after conversion by the schema) is **identical** to what was saved, including the `healthy` column that holds a `bool`. The JSON is identical too. Now test its resilience:

```
# Chapter 14 - Project B3: loading a CSV that field staff have messed up, plus missing
or broken files
import logging
import sys
from pathlib import Path
```

```
from data_tools import save_csv, load_csv, load_json
from core_data import crops

logging.basicConfig(stream=sys.stdout, level=logging.WARNING, format="%(levelname)s:
%(message)s")

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

save_csv(file, crops)
with open(file, "a", newline="", encoding="utf-8") as f:           # what the field
staff did
    f.write("rice,E,hundred,250.0\n")                             # age is not a number
    f.write("maize,F,95\n")                                        # missing column

schema = {"crop": str, "block": str, "age_days": int, "yield_kg": float}
rows, rejected = load_csv(file, schema)
print(f"Accepted {len(rows)} rows, rejected {rejected}")
print("Total valid harvest:", sum(r["yield_kg"] for r in rows), "kg")

print("Missing file:", load_csv(tmp / "does_not_exist.csv", schema))

(tmp / "broken.json").write_text('{"trees": [1, 2,', encoding="utf-8")
print("Broken JSON -> fallback:", load_json(tmp / "broken.json", fallback={}))
```

```
WARNING: line 6 in crops_dirty.csv rejected (ValueError)
WARNING: line 7 in crops_dirty.csv rejected (TypeError)
Accepted 4 rows, rejected 2
Total valid harvest: 1030.0 kg
WARNING: file does_not_exist.csv does not exist yet
Missing file: ([], 0)
WARNING: file broken.json is not valid JSON
Broken JSON -> fallback: {}
```

The staff added two broken rows: `hundred` cannot become an `int` (`ValueError`), and a row with a missing column makes `DictReader` fill in `None`, which fails to convert (`TypeError`). Both are **rejected with a warning**, the four valid rows are still read (total 1030.0 kg, matching Chapter 11). A missing file gives an empty list, and a broken JSON gives the fallback value `{}`. None of it collapses the program: that is what "resistant to wrong input" means.

Exercises

1. Write the `livestock` data to `livestock.csv`, read it back with `DictReader`, and print the total `weight_kg` (don't forget to convert the types).
2. Save the dict `{"name": "Pond Ø1", "temp": "28 °C"}` to JSON with and without `ensure_ascii=False`. Compare the file contents.
3. Change `08_project_save.py` so that the tree `height_m` is rounded to a whole number when loaded (change the schema, not the data).
4. **Challenge:** write a function `compare_csv_json(name)` that loads table `name` from CSV and from JSON, then reports whether they are the same. What happens if a `True` value is stored via CSV without the `bool_text` converter?

Chapter Summary

- **Serialisation** = turning data in memory into text/bytes for storage; **deserialisation** = the reverse.
- **CSV**: flat table; open with `newline=""`; all values arrive as **text**, convert the types yourself.
- Don't cut CSV with `split(",")`; use the `csv` module.

- **JSON**: nested data; gives back `bool`, numbers, `None`, list, dict; knows no tuples or dates.
- **Pickle is not secure**: opening one can run code. Never open a pickle from a source you don't trust.
- Project B3: five sectors saved in CSV/JSON and loaded back with a schema, resistant to broken rows, missing files, and broken JSON.

Next, in Chapter 15: office document automation, turning that data into Excel, Word, and PDF reports.

Chapter 15 — Office Document Automation: From Data to Excel and PDF Reports

Series B3 — Small Programs That Are Useful

In this chapter, you will:

- Make and read **Excel** files with `openpyxl` and `pandas`
- Make and read **Word** files with `python-docx`
- Make **PDFs** with `reportlab`, read and merge them with `pypdf`, and pull tables out with `pdfplumber`
- Build an **email** object with attachments, and test it against a pretend local SMTP server (without sending real email)
- Finish the Series B3 project: turning field data into Excel and PDF reports

All scripts for this chapter are in the `kode/ch15/` folder. On the web version, you can also download them all at once (zip). Result files are written to the `_tmp/` subfolder.

Every weekend, someone in the field office copies numbers from a table into Excel, formats them, turns them into a PDF, and pastes it into an email. The job is boring, error-prone, and takes the same time every week. Python can do it in seconds. That is **office document automation**: a program makes the documents, instead of a human retyping them.

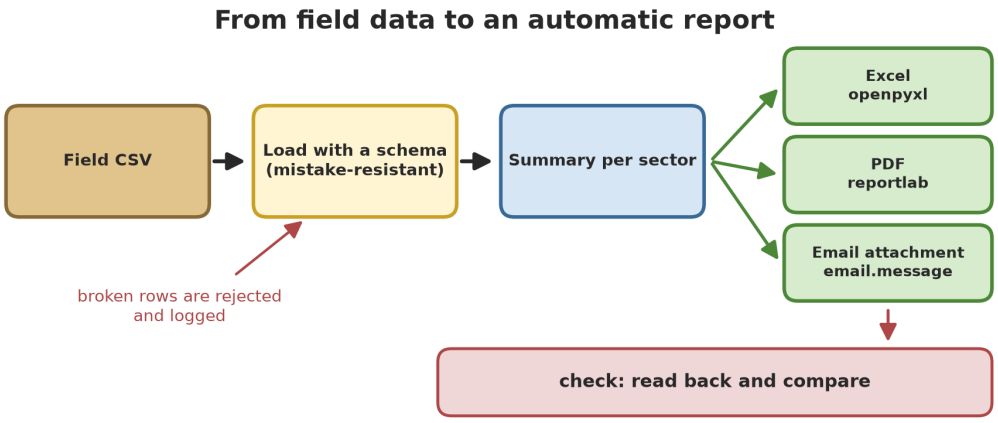


Figure 0.1: The project flow of this chapter: field CSV is loaded resistant to mistakes, summarised, then comes out as Excel, PDF, and an email attachment.

Installing the libraries

This chapter uses libraries outside the standard library (Chapter 11). The installation command (typed in the terminal) is only an example and was **not run** when this book was built, because installing packages changes the reader's computer:

```
python -m pip install openpyxl pandas python-docx pypdf reportlab pdfplumber aiosmtpd
```

Library	What it does in this chapter
`openpyxl`	reads and writes Excel `.xlsx` files (sheets, styles, formulas, charts)
`pandas`	handles tables, then reads and writes Excel in one line
`python-docx`	makes and reads Word `.docx` files
`reportlab`	**makes** PDFs from scratch
`pypdf`	**reads** PDF text, and merges and splits pages
`pdfplumber`	pulls **tables** out of PDFs
`aiosmtpd`	a pretend SMTP server to test email sending locally

We ran all of them on the author's computer (Python 3.14) before writing about them here; the output in the book is real. None of these libraries needs Microsoft Office; the files they produce can be opened in Excel or LibreOffice.

Excel with openpyxl

```
# Chapter 15 - Excel with openpyxl: making a workbook, bold titles, numbers, and
a formula
from pathlib import Path

from openpyxl import Workbook
from openpyxl.styles import Font, PatternFill

from core_data import crops

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

book = Workbook()
sheet = book.active
sheet.title = "Harvest"

sheet.append(["Crop", "Block", "Age (days)", "Yield (kg)"])         # title row
for c in crops:
    sheet.append([c["crop"], c["block"], c["age_days"], c["yield_kg"]])

for cell in sheet[1]:                                               # style of the
title row
    cell.font = Font(bold=True)
    cell.fill = PatternFill("solid", start_color="D7ECCF")
sheet.column_dimensions["A"].width = 14

last = sheet.max_row
sheet[f"C{last + 1}"] = "Total"
sheet[f"D{last + 1}"] = f"=SUM(D2:D{last})"                         # an Excel formula
as text
sheet[f"D{last + 1}"].number_format = "#,##0.0"

book.save(tmp / "harvest.xlsx")
print("Filled rows:", sheet.max_row, "| columns:", sheet.max_column)
print("Content of cell D6:", sheet["D6"].value)
```

```
Filled rows: 6 | columns: 4
Content of cell D6: =SUM(D2:D5)
```

- `Workbook()` makes a new workbook; `.active` is its first sheet.
- `sheet.append([...])` adds one row below the last row.
- `Font(bold=True)` and `PatternFill(...)` style the title row.
- An Excel formula is written as text starting with `=`. The data rows are 2 to 5, so the total goes in row 6, cell `D6`.

In cell `D6`, Python only **stores the formula text**. Who calculates it? Excel or LibreOffice, when the file is opened. We proved it by converting that file to CSV through LibreOffice: its last row holds the total 1030 (same as the result of Chapter 11).

```
# Chapter 15 - Reading Excel: a formula value exists only if Excel has calculated it
from pathlib import Path

from openpyxl import Workbook, load_workbook

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

book = Workbook()
ws = book.active
ws.append(["block", "yield_kg"])
for row in [["A", 245.5], ["B", 310.0], ["C", 198.5]]:
    ws.append(row)
ws["B5"] = "=SUM(B2:B4)"
book.save(file)

opened = load_workbook(file)                              # default: formulas are read
as text
print("Sheet names:", opened.sheetnames)
for row in opened.active.iter_rows(min_row=2, values_only=True):
    print(row)

print("Formula   :", opened.active["B5"].value)
values = load_workbook(file, data_only=True)        # ask for the LAST CALCULATED result
print("Its result:", values.active["B5"].value)          # None: this file was never
opened in Excel
```

```
Sheet names: ['Sheet']
('A', 245.5)
('B', 310)
('C', 198.5)
(None, '=SUM(B2:B4)')
Formula   : =SUM(B2:B4)
Its result: None
```

Opened normally, cell `B5` holds the **formula text**. Opened with `data_only=True`, it gives the last calculated result stored by Excel; the openpyxl documentation describes this as "the value stored the last time Excel read the sheet". Because this file was just made by Python and has never been opened in Excel, the result is `None`. Another small thing to know: `310.0` is read back as `310` (a whole number), because Excel does not distinguish the two.

> ⚠ WARNING
>
> `openpyxl` **does not calculate formulas**. If what you need is the numbers (for another program, or to put in a PDF), calculate in Python and write the numbers. Write a formula only if the reader in Excel needs to see or change it.

Charts inside Excel

```
# Chapter 15 - A summary sheet with a built-in Excel bar chart (openpyxl.chart)
import zipfile
from pathlib import Path

from openpyxl import Workbook
from openpyxl.chart import BarChart, Reference

from core_data import crops, fish, livestock

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

total_harvest = sum(c["yield_kg"] for c in crops)
total_fish = sum(f["count"] * f["avg_weight_g"] / 1000 for f in fish)
total_livestock = sum(x["weight_kg"] for x in livestock)

book = Workbook()
ws = book.active
ws.title = "Summary"
ws.append(["Sector", "Total (kg)"])
ws.append(["Orchard and field harvest", total_harvest])
ws.append(["Fish biomass", round(total_fish, 1)])
ws.append(["Livestock weight", total_livestock])

chart = BarChart()
chart.title = "Total per sector"
chart.y_axis.scaling.min = 0                    # axis starts at zero to stay honest
chart.add_data(Reference(ws, min_col=2, min_row=1, max_row=4), titles_from_data=True)
chart.set_categories(Reference(ws, min_col=1, min_row=2, max_row=4))
ws.add_chart(chart, "D2")
book.save(file)

with zipfile.ZipFile(file) as z:                    # an .xlsx is really a zip full
of XML
    print("Chart files inside the xlsx:", [n for n in z.namelist() if "chart" in n
and n.endswith(".xml")])
print("Number of charts on the sheet:", len(ws._charts))
print("Table content:", [tuple(r) for r in ws.iter_rows(min_row=2, values_only=True)])
```

```
Chart files inside the xlsx: ['xl/charts/chart1.xml']
Number of charts on the sheet: 1
Table content: [('Orchard and field harvest', 1030.0), ('Fish biomass', 1095.4),
('Livestock weight', 1190.0)]
```

The bar chart is made from `BarChart` and two `Reference` objects (the data range and the label range). `scaling.min = 0` forces the axis to start at zero: without it, Excel may choose an axis that starts at 950 so the bars look wildly different though the gap is small. An `.xlsx` file is really a **zip full of XML**; the output shows `xl/charts/chart1.xml` inside it.

Excel with pandas

For **tables**, `pandas` is more concise. One `DataFrame` is a table in memory:

```
# Chapter 15 - pandas: CSV to summary to Excel (several sheets) and back
from pathlib import Path

import pandas as pd
```

```
from core_data import trees, fish

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

df_trees = pd.DataFrame(trees)
df_fish = pd.DataFrame(fish)
df_fish["biomass_kg"] = df_fish["count"] * df_fish["avg_weight_g"] / 1000

plot_summary          =          df_trees.groupby("plot")["diameter_cm"].agg(["count",
"mean"]).round(2)
print(plot_summary.to_string())

file = tmp / "pandas_summary.xlsx"
with pd.ExcelWriter(file, engine="openpyxl") as writer:         # one file, many sheets
    df_trees.to_excel(writer, sheet_name="Trees", index=False)
    df_fish.to_excel(writer, sheet_name="Fish", index=False)
    plot_summary.to_excel(writer, sheet_name="Plot summary")

back = pd.read_excel(file, sheet_name=None)                      # None = all sheets
at once
print("Sheets:", list(back))
print("Total fish biomass:", back["Fish"]["biomass_kg"].sum(), "kg")
print("Type of healthy column:", back["Trees"]["healthy"].dtype)
```

```
      count   mean
plot
A         2  38.85
B         2  39.75
C         2  38.40
Sheets: ['Trees', 'Fish', 'Plot summary']
Total fish biomass: 1095.4 kg
Type of healthy column: bool
```

- `DataFrame(trees)` turns a list of dicts into a table.
- `groupby("plot")` groups per plot, `agg(["count", "mean"])` calculates the count and mean. Plot A: 2 trees, mean 38.85 cm.
- `pd.ExcelWriter` writes several sheets in one file; `read_excel(sheet_name=None)` reads all sheets at once as a dict.
- The `healthy` column comes back as a `bool`, and the total biomass is 1095.4 kg (432.4 + 375.0 + 288.0). (Pandas is covered fully in Series I2.)

Word with python-docx

```
# Chapter 15 - Word with python-docx: heading, paragraph, table, then read back
from pathlib import Path

from docx import Document

from core_data import fish

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

doc = Document()
doc.add_heading("Pond Report (sample data)", level=1)
```

```
doc.add_paragraph("Biomass summary of three ponds in early October. Fictional numbers
for practice.")

table = doc.add_table(rows=1, cols=3)
table.style = "Table Grid"
for cell, title in zip(table.rows[0].cells, ["Pond", "Species", "Biomass (kg)"]):
    cell.text = title
for f in fish:
    row = table.add_row().cells
    row[0].text = f["pond"]
    row[1].text = f["species"]
    row[2].text = f"{f['count'] * f['avg_weight_g'] / 1000:.1f}"
doc.save(file)

reopened = Document(file)                              # open again and check
print("Paragraphs:", [p.text for p in reopened.paragraphs])
print("Number      of      tables:",      len(reopened.tables),      "|      rows:",
len(reopened.tables[0].rows))
for row in reopened.tables[0].rows:
    print([cell.text for cell in row.cells])
```

```
Paragraphs: ['Pond Report (sample data)', 'Biomass summary of three ponds in early
October. Fictional numbers for practice.']
Number of tables: 1 | rows: 4
['Pond', 'Species', 'Biomass (kg)']
['A', 'catfish', '432.4']
['B', 'tilapia', '375.0']
['C', 'catfish', '288.0']
```

`Document()` makes an empty document; `add_heading`, `add_paragraph`, and `add_table` add its contents. After saving, we **open it again** and read the paragraphs and table cells: that is the proof the file isn't empty. The `"Table Grid"` style (a table with lines) exists in python-docx's default template; we tested it directly.

PDF: making with reportlab

A PDF is designed to be **read**, not edited. To make one from Python, `reportlab` offers "platypus": you build a list of elements (heading, paragraph, table), then `build`. The basic maker is in the module `report_pdf.py` (see the folder `kode/ch15/`).

```
# Chapter 15 - PDF with reportlab: making a one-page report
from pathlib import Path

from report_pdf import make_pdf
from core_data import livestock

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

table = [["Pen", "Species", "Age (months)", "Weight (kg)"]]
for x in livestock:
                    table.append([x["pen"],     x["species"],     str(x["age_months"]),
f"{x['weight_kg']:.1f}"])
make_pdf(file, "Livestock Report", table, "Livestock weight per pen (sample numbers).")

print("PDF made:", file.name, "| start of file:", file.read_bytes()[:8])
```

```
PDF made: livestock.pdf | start of file: b'%PDF-1.4'
```

The file start `%PDF-1.4` is the identification mark of the PDF format.

PDF: reading and pulling out tables

```
# Chapter 15 - Reading PDF: text with pypdf, tables with pdfplumber
from pathlib import Path

import pdfplumber
from pypdf import PdfReader

from report_pdf import make_pdf
from core_data import livestock

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
file = tmp / "livestock_read.pdf"
table = [["Pen", "Species", "Age (months)", "Weight (kg)"]]
table += [[x["pen"], x["species"], str(x["age_months"]), f"{x['weight_kg']:.1f}"] for
x in livestock]
make_pdf(file, "Livestock Report", table, "Livestock weight per pen (sample numbers).")

reader = PdfReader(file)
print("Number of pages:", len(reader.pages))
print("Metadata title :", reader.metadata.title)
print("--- text (pypdf) ---")
text_lines = reader.pages[0].extract_text().splitlines()
print(len(text_lines), "lines; first 7:", text_lines[:7])

print("--- table (pdfplumber) ---")
with pdfplumber.open(file) as pdf:
    for row in pdf.pages[0].extract_tables()[0]:
        print(row)
```

```
Number of pages: 1
Metadata title : Livestock Report
--- text (pypdf) ---
22 lines; first 7: ['Livestock Report', 'Livestock weight per pen (sample numbers).',
'Pen', 'Species', 'Age (months)', 'Weight (kg)', 'K1']
--- table (pdfplumber) ---
['Pen', 'Species', 'Age (months)', 'Weight (kg)']
['K1', 'cattle', '36', '350.0']
['K1', 'cattle', '30', '412.5']
['K2', 'cattle', '42', '389.0']
['K3', 'goat', '14', '38.5']
```

Two tools, two different results:

- **`pypdf`** takes the **text**, but the table structure is lost: each cell becomes one line (22 lines, with cells appearing in sequence, `Pen`, `Species`, `Age (months)`, ...).
- **`pdfplumber`** recognises the table lines and returns each row as a list of cells: five neat rows.

An honest note from the pdfplumber documentation: it works best on **machine-generated** PDFs (like the ones we make), not **scanned** PDFs (photos of paper). Scanned PDFs need OCR, which this book does not cover. For PDFs without table lines, the result of `extract_tables()` may be less accurate; always check.

PDF: merging and splitting

```
# Chapter 15 - Merging and splitting PDFs with pypdf
from pathlib import Path
```

```
from pypdf import PdfReader, PdfWriter

from report_pdf import make_pdf

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

make_pdf(tmp / "part_forest.pdf", "Forest Part", [["Plot", "Trees"], ["A", "2"], ["B",
"2"]])
make_pdf(tmp / "part_ponds.pdf", "Pond Part", [["Pond", "kg"], ["A", "432.4"], ["B",
"375.0"]])

writer = PdfWriter()
for name in ["part_forest.pdf", "part_ponds.pdf"]:
    writer.append(str(tmp / name))                    # add all pages of that file
writer.add_metadata({"/Title": "Combined Report"})
with open(tmp / "combined.pdf", "wb") as f:
    writer.write(f)

combined = PdfReader(tmp / "combined.pdf")
print("Combined pages:", len(combined.pages))
for no, page in enumerate(combined.pages, start=1):
    print(f"  page {no}:", page.extract_text().splitlines()[0])

single = PdfWriter()                                  # splitting: take only page 2
single.add_page(combined.pages[1])
with open(tmp / "page2.pdf", "wb") as f:
    single.write(f)
print("Pages after splitting:", len(PdfReader(tmp / "page2.pdf").pages))
```

```
Combined pages: 2
  page 1: Forest Part
  page 2: Pond Part
Pages after splitting: 1
```

`PdfWriter.append()` adds all pages of another file; two one-page PDFs become one two-page PDF. Splitting is done by choosing a page (`pages[1]` is the second page, because numbering starts at 0) and writing it as a new file.

Email: building a message

Python has the `email.message` module to **compose** email, including attachments:

```
# Chapter 15 - Making an email object with email.message (NOT sent anywhere)
from email.message import EmailMessage
from pathlib import Path

from report_pdf import make_pdf

tmp = Path(__file__).parent / "_tmp"
tmp.mkdir(exist_ok=True)
attachment = tmp / "summary.pdf"
make_pdf(attachment, "Weekly Summary", [["Sector", "Total"], ["Harvest (kg)",
"1030.0"]])

message = EmailMessage()
message["Subject"] = "Weekly field data summary"
message["From"] = "reporting@sample.example"        # fictional address (.example is
a reserved domain)
```

```
message["To"] = "unit-head@sample.example"
message.set_content("Good morning,\n\nThe field data summary for this week is attached.
\n\nRegards,\nField Team")
message.add_attachment(attachment.read_bytes(),
maintype="application", subtype="pdf",
                       filename="weekly_summary.pdf")

print("Subject   :", message["Subject"])
print("From / To :", message["From"], "->", message["To"])
print("Multipart?", message.is_multipart())
print("Text                                  body                                    :",
message.get_body(preferencelist=("plain",)).get_content().splitlines()[0])
for att in message.iter_attachments():
    print("Attachment:", att.get_filename(), att.get_content_type())
print("This email is only an object in memory; nothing has been sent.")
```

```
Subject   : Weekly field data summary
From / To : reporting@sample.example -> unit-head@sample.example
Multipart? True
Text body : Good morning,
Attachment: weekly_summary.pdf application/pdf
This email is only an object in memory; nothing has been sent.
```

This message is an **object in memory**; nothing has been sent. The addresses are fictional, using the `.example` domain that is reserved for examples. A message with an attachment automatically becomes *multipart* (`True`).

> ⚠ WARNING
>
> This book **does not send real email** and we ask you to be careful: an automatic email sent to the wrong address, or repeated out of control, can leak data or flood someone's inbox. Always test against a pretend server first, and never write passwords in code (take them from environment variables).

Email: testing against a local server

A safe way to test sending: run a **pretend SMTP server on your own computer**, and send to it. We use `aiosmtpd`; its documentation says the `Controller` runs an SMTP server in a separate thread, which suits setting up a server in tests.

```
# Chapter 15 - Sending email to a pretend LOCAL SMTP server (only on this computer,
not the internet)
import smtplib
import socket
from email.message import EmailMessage

from aiosmtpd.controller import Controller


class Collector:
    """Handler of the pretend server: keeps every message that arrives."""
    def __init__(self):
        self.mails = []

    async def handle_DATA(self, server, session, envelope):
        self.mails.append(envelope)
        return "250 Message accepted (pretend server)"


def free_port():
```

```
    """Asks the operating system to pick one empty port on this computer."""
    with socket.socket() as s:
        s.bind(("127.0.0.1", 0))
        return s.getsockname()[1]


collector = Collector()
port = free_port()
server = Controller(collector, hostname="127.0.0.1", port=port)   # accepts only from
this computer
server.start()
try:
    message = EmailMessage()
    message["Subject"] = "Local sending test"
    message["From"] = "reporting@sample.example"
    message["To"] = "unit-head@sample.example"
   message.set_content("This is only a test to a pretend server on our own computer.")

    with smtplib.SMTP("127.0.0.1", port) as client:
        client.send_message(message)
finally:
    server.stop()

received = collector.mails[0]
print("Mails received by the pretend server:", len(collector.mails))
print("Sender   :", received.mail_from)
print("Receiver :", received.rcpt_tos)
print("Test subject present?", b"Subject: Local sending test" in received.content)
```

```
Mails received by the pretend server: 1
Sender   : reporting@sample.example
Receiver : ['unit-head@sample.example']
Test subject present? True
```

The server only listens on `127.0.0.1` (the address meaning "this computer itself"), so the message never leaves for the internet. The pretend server receives one message, and the sender, receiver, and subject match what we sent. To send for real, you use `smtplib.SMTP_SSL` with the server address, user name, and password from your email provider. It looks like this (**not run** here, because it would send real email):

```
import os, smtplib, ssl
with                  smtplib.SMTP_SSL(os.environ["SMTP_HOST"],                  465,
context=ssl.create_default_context()) as s:
       s.login(os.environ["SMTP_USER"],  os.environ["SMTP_PASSWORD"])      #  values:
[disamarkan]
    s.send_message(message)
```

Running Project B3: field summary becomes a report

Now the whole of Series B3 works together. The script below: (1) saves the data of five sectors to CSV and **deliberately adds one broken row** to the fish data, (2) loads it back with a schema (Chapter 14), (3) works out a summary per sector, (4) makes an **Excel** report (Summary sheet + chart + one sheet per table), (5) makes a **PDF** report, and (6) builds an email with two attachments (not sent).

```
# Chapter 15 - Project B3: from field CSV to an Excel and PDF report, plus an email
(not sent)
import logging
import sys
from email.message import EmailMessage
from pathlib import Path
```

```
from openpyxl import Workbook
from openpyxl.chart import BarChart, Reference
from openpyxl.styles import Font, PatternFill

from data_tools import save_csv, load_csv
from core_data import trees, crops, fish, livestock
from report_pdf import make_pdf

logging.basicConfig(stream=sys.stdout, level=logging.WARNING, format="%(levelname)s:
%(message)s")
work = Path(__file__).parent / "_tmp" / "project"
work.mkdir(parents=True, exist_ok=True)

# 1. Save the raw data to CSV; a field officer adds one broken row to the fish data
tables = {"trees": trees, "crops": crops, "fish": fish, "livestock": livestock}
for name, rows in tables.items():
    save_csv(work / f"{name}.csv", rows)
with open(work / "fish.csv", "a", newline="", encoding="utf-8") as f:
    f.write("D,tilapia,many,350.0,7.4\n")

# 2. Load it back with a schema (the broken row is rejected)
bool_text = lambda s: {"True": True, "False": False}[s]
schema = {
     "trees": {"species": str, "diameter_cm": float, "height_m": float, "healthy":
bool_text, "plot": str},
    "crops": {"crop": str, "block": str, "age_days": int, "yield_kg": float},
      "fish": {"pond": str, "species": str, "count": int, "avg_weight_g": float,
"water_ph": float},
    "livestock": {"pen": str, "species": str, "age_months": int, "weight_kg": float},
}
data, rejected = {}, 0
for name in tables:
    data[name], rej = load_csv(work / f"{name}.csv", schema[name])
    rejected += rej

# 3. Work out the summary per sector
biomass = sum(f["count"] * f["avg_weight_g"] / 1000 for f in data["fish"])
summary = [
    ["Sector", "Records", "Measure", "Value"],
    ["Forest", len(data["trees"]), "Mean diameter (cm)",
     round(sum(t["diameter_cm"] for t in data["trees"]) / len(data["trees"]), 1)],
       ["Forest", len(data["trees"]), "Healthy trees", sum(t["healthy"] for t in
data["trees"])],
    ["Orchard and field", len(data["crops"]), "Total harvest (kg)", sum(c["yield_kg"]
for c in data["crops"])],
    ["Fisheries", len(data["fish"]), "Biomass (kg)", round(biomass, 1)],
    ["Livestock", len(data["livestock"]), "Total weight (kg)", sum(x["weight_kg"] for
x in data["livestock"])],
]

# 4. Excel report: Summary sheet + chart + one sheet per table
book = Workbook()
ws = book.active
ws.title = "Summary"
for row in summary:
    ws.append(row)
ws.append([])
ws.append([f"Data rows rejected because they were broken: {rejected}"])
for cell in ws[1]:
```

```
    cell.font = Font(bold=True)
    cell.fill = PatternFill("solid", start_color="D7ECCF")
ws.column_dimensions["A"].width = 18
ws.column_dimensions["C"].width = 26
chart = BarChart()
chart.title = "Value per measure (kg)"
chart.y_axis.scaling.min = 0
chart.add_data(Reference(ws, min_col=4, min_row=4, max_row=6), titles_from_data=False)
chart.set_categories(Reference(ws, min_col=3, min_row=4, max_row=6))
ws.add_chart(chart, "F2")
for name, rows in data.items():
    sheet = book.create_sheet(name.capitalize())
    sheet.append(list(rows[0]))
    for r in rows:
        sheet.append(list(r.values()))
book.save(work / "report.xlsx")

# 5. PDF report
pdf_rows = [summary[0]] + [[s, str(n), m, str(v)] for s, n, m, v in summary[1:]]
make_pdf(work / "report.pdf", "Field Inventory Summary", pdf_rows,
         f"Sample numbers. Data rows rejected because they were broken: {rejected}.")

# 6. Email with two attachments (only made, not sent)
message = EmailMessage()
message["Subject"] = "Field inventory summary"
message["From"] = "reporting@sample.example"
message["To"] = "unit-head@sample.example"
message.set_content("The inventory summary is attached in Excel and PDF.")
message.add_attachment((work / "report.xlsx").read_bytes(), maintype="application",
                       subtype="vnd.openxmlformats-officedocument.spreadsheetml.sheet",
filename="report.xlsx")
message.add_attachment((work / "report.pdf").read_bytes(), maintype="application",
                       subtype="pdf", filename="report.pdf")

print(f"Rows rejected: {rejected}")
for row in summary[1:]:
    print(f"{row[0]:18} {row[1]:>2}  {row[2]:24} {row[3]}")
print("Report files:", sorted(p.name for p in work.iterdir() if p.suffix in (".xlsx",
".pdf")))
print("Email attachments:", [a.get_filename() for a in message.iter_attachments()],
"(not sent yet)")
```

```
WARNING: line 5 in fish.csv rejected (ValueError)
Rows rejected: 1
Forest              6  Mean diameter (cm)       39.0
Forest              6  Healthy trees            5
Orchard and field   4  Total harvest (kg)       1030.0
Fisheries           3  Biomass (kg)             1095.4
Livestock           4  Total weight (kg)        1190.0
Report files: ['report.pdf', 'report.xlsx']
Email attachments: ['report.xlsx', 'report.pdf'] (not sent yet)
```

Follow the result:

- The broken row `D,tilapia,many,...` is rejected (the count is `many`, not a number), and the report records **rows rejected: 1**. The remaining three ponds are still processed: biomass 1095.4 kg.
- The per-sector summary matches Chapter 11: harvest 1030.0 kg, livestock weight 1190.0 kg, mean diameter 39.0 cm; 5 of 6 trees healthy.
- The files `report.xlsx` and `report.pdf` are made, and the email with both attached is ready, not sent. We also opened `report.xlsx` with LibreOffice (converting it to PDF) to confirm the file is valid and its chart shows up.

An automatic program must not be trusted blindly. The next script **reads back** the generated reports and compares Excel with PDF:

```
# Chapter 15 - Project B3: checking the generated reports by reading them back (proof,
not assumption)
import subprocess
import sys
from pathlib import Path

import pdfplumber
from openpyxl import load_workbook
from pypdf import PdfReader

folder = Path(__file__).parent
work = folder / "_tmp" / "project"
subprocess.run([sys.executable,      str(folder      /      "11_project_report.py")],
capture_output=True, check=True)

book = load_workbook(work / "report.xlsx")
print("Excel sheets:", book.sheetnames)
rows = list(book["Summary"].iter_rows(min_row=2, max_row=6, values_only=True))
excel_totals = {row[2]: row[3] for row in rows}
print("Excel - total harvest:", excel_totals["Total harvest (kg)"])
print("Excel - biomass      :", excel_totals["Biomass (kg)"])
print("Excel - fish rows    :", book["Fish"].max_row - 1, "(without the title row)")

reader = PdfReader(work / "report.pdf")
print("PDF   - pages        :", len(reader.pages))
with pdfplumber.open(work / "report.pdf") as pdf:
    table = pdf.pages[0].extract_tables()[0]
pdf_totals = {row[2]: row[3] for row in table[1:]}
print("PDF   - total harvest:", pdf_totals["Total harvest (kg)"])
print("PDF   - biomass      :", pdf_totals["Biomass (kg)"])

# Excel stores 1030.0 as 1030, so compare as numbers, not as text
same = all(float(excel_totals[k]) == float(pdf_totals[k]) for k in ("Total harvest
(kg)", "Biomass (kg)"))
print("Excel and PDF consistent:", same)
```

```
Excel sheets: ['Summary', 'Trees', 'Crops', 'Fish', 'Livestock']
Excel - total harvest: 1030
Excel - biomass      : 1095.4
Excel - fish rows    : 3 (without the title row)
PDF   - pages        : 1
PDF   - total harvest: 1030.0
PDF   - biomass      : 1095.4
Excel and PDF consistent: True
```

Excel has five sheets, the fish table holds 3 data rows, and the totals in Excel and PDF are the same. Note one small lesson: Excel stores the total harvest as `1030`, while the PDF has the text `1030.0`. Comparing the two as **text** would report "different" though the numbers are equal; we compare them as **numbers** (`float`). A check like this is what lets you dare to schedule a report to run by itself.

Exercises

1. Change `01_excel_write.py`: add a column "Yield per 100 days (kg)" filled by Python (not a formula), namely `yield_kg / age_days * 100`, rounded to two decimals.
2. Use `pandas` to read the `fish.csv` from Chapter 14 and write `fish.xlsx` with one sheet named "Fish". Open it with `read_excel` and print the mean of `avg_weight_g`.
3. Change `11_project_report.py` so that the PDF report also holds a summary table **per forest plot** (number of trees and mean diameter).

4. **Challenge:** write a script that merges the project's `report.pdf` with a cover PDF you make yourself (one title page) into `full_report.pdf`, then check with `pypdf` that the first page is the cover.

Chapter Summary

- **Excel**: `openpyxl` (sheets, styles, formulas, charts) and `pandas` (tables); openpyxl does not calculate formulas.
- **Word**: `python-docx` for headings, paragraphs, and tables.
- **PDF**: `reportlab` makes; `pypdf` reads text, merges, and splits; `pdfplumber` pulls tables from machine-generated PDFs.
- **Email**: `email.message.EmailMessage` builds the message; test against a pretend local SMTP server; never write passwords in code.
- An automatic report needs an **automatic check**: read the result back and compare.
- The complete B3 project: data of five sectors saved, resistant to wrong input, and turned into Excel/PDF reports.

Closing Series B3

You have finished Series B3. Your program can now **borrow tools** from libraries (Chapter 11), **stay up** when something goes wrong (Chapter 12), **save and read files** (Chapter 13), **use the standard formats CSV and JSON** (Chapter 14), and **make Excel and PDF reports automatically** (Chapter 15). You also have one complete project: data of forest, orchard, field crops, fisheries, and livestock that is saved, mistake-resistant, and turned into reports.

Congratulations: Basic Level (Volume 1) is complete!

With this chapter, the **Basic level is finished**. In 15 chapters you travelled from "what is a computer" to writing programs that are useful at work: variables, decisions, loops, data structures, functions, modules, error handling, files, data formats, and document automation. That is a genuine foundation, and many everyday office jobs can already be automated with it.

But see for yourself: our project program is getting bigger, its functions are multiplying, and it is starting to get **messy**: data and tools are mixed together, and there are copies of the file `core_data.py` in several chapter folders. **The Intermediate level begins with how to organise code** so that it stays tidy and safe to change.

Next, in Chapter 16: OOP (object-oriented programming): classes, objects, inheritance, and how to model trees, orchards, ponds, and pens as objects that have their own behaviour.

Appendix A — Installing on Windows, Linux, and macOS

In this chapter, you will:

- Install Python 3.14 on Windows, Linux, or macOS (or use `uv` on all three)
- Create and activate a virtual environment (**venv**) on each system
- Check that your installation is healthy, using a small script
- Install the supporting tools this book uses: VS Code and Git
- Recognise the most common installation problems and how to fix them

All scripts of this appendix are in the folder `kode/ch71/`. In the web version you can also download them all at once (zip).

Chapter 4 already walked you through installing Python on Windows. This appendix completes it: it gathers **all operating systems** in one place, adds how to create virtual environments and supporting tools, and ends with a "if this happens, try that" list. Think of it as the manual in the workshop drawer: you do not read it from start to finish, you open it when a bolt is stuck.

Honesty first: what was tested and what was not

This book was written and run on **one Windows 11 computer**. Therefore:

Part	Status	Basis
Windows (Python install manager, `py`, venv, `uv`)	**Tested** on the author's computer	Commands were run; output is shown
Linux (distribution packages, pyenv, uv)	**Not tested on Linux**	Official documentation, read 2026-10-10
macOS (python.org installer, Homebrew, uv)	**Not tested on macOS**	Official documentation, read 2026-10-10

Every Linux and macOS command below is marked with its source. When I could not find a command in the official documentation, I write **not verified** instead of presenting it as fact. Version numbers on your computer may differ from those shown here.

The terminal: where you type commands

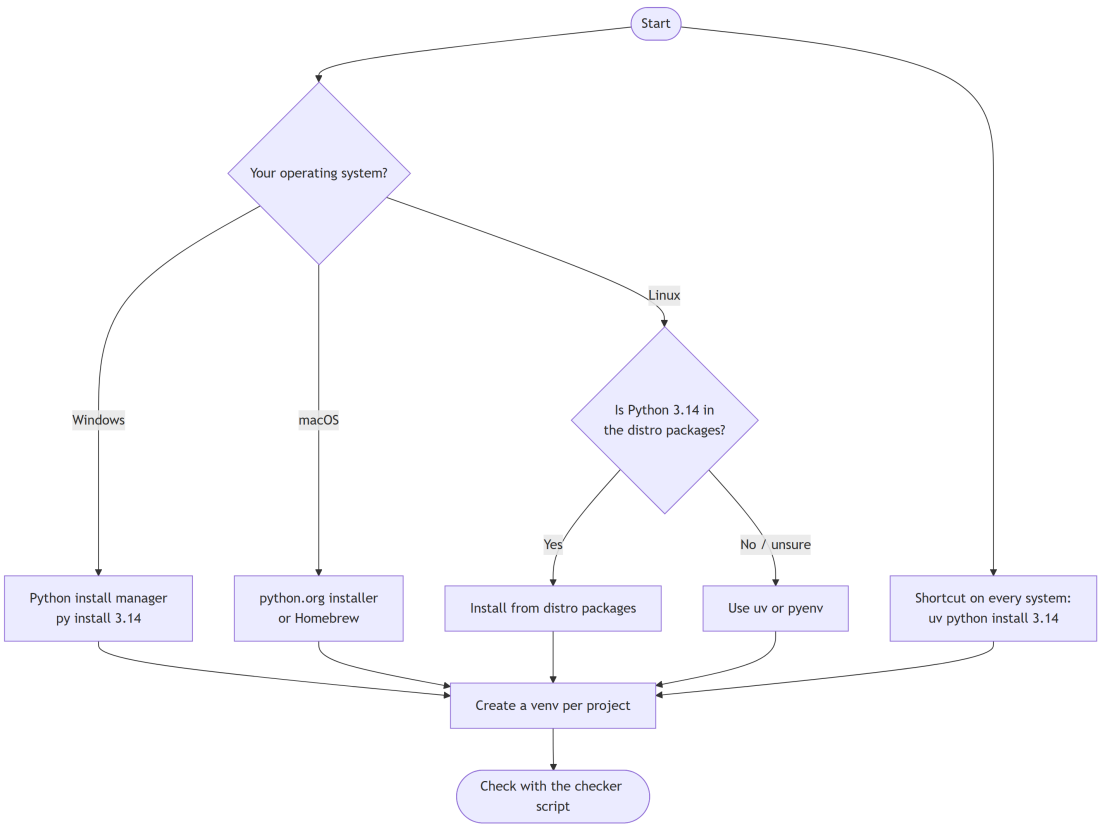
All commands in this appendix are typed in a **terminal** (explained in Chapter 3):

- **Windows:** open *PowerShell* (search for "PowerShell" in the Start menu).
- **macOS:** open the *Terminal* app (in Launchpad, folder Other).
- **Linux:** open the *Terminal* app from your application menu.

The `$` at the start of Linux and macOS example lines is just the "ready for a command" sign (the *prompt*). Do not type it.

> **i** REMEMBER
>
> Python has two confusing "names": `python` and `python3`. On Windows with the Python install manager, both `python` and `py` work. On Linux and macOS, get used to typing **`python3`**, because `python` is sometimes missing or points to another version.

Decision map: which way to install?



Choosing how to install Python: the main route per system, and uv as a shortcut on all of them. Trace it from the oval **Start**. Almost every route ends at the same step: **create a venv per project** (section below), then check the result.

Windows

The Python install manager

The official Python 3.14 documentation for Windows (docs.python.org/3.14/using/windows.html, read 2026-10-10) names the **Python install manager** as the recommended way. The old single-file installer has been deprecated since 3.14 and will not be produced for Python 3.16 and later. There are three ways to install the manager itself:

1. **Microsoft Store:** find the Python install manager app and click Install.
2. **MSIX file from python.org:** download it from python.org/downloads, double-click, choose Install. The Store and python.org packages are identical, but **cannot be installed at the same time**.
3. **MSI** (for special cases, for example Windows Server 2019): no user interface, installs for the whole computer. Beginners do not need this.

Once the manager is installed, install Python itself:

```
py install 3.14
```

Other manager commands listed in the same documentation:

Command	What it does
`py list`	Shows the Python runtimes that are installed
`py list --online 3.14`	Shows what can be installed
`py install 3.14`	Installs Python 3.14

Command	What it does
`py uninstall 3.14`	Removes Python 3.14
`py -V:3.14 script.py`	Runs a script with Python 3.14 (this option must come first)
`py install --refresh`	Recreates Start menu shortcuts and aliases

Results on the author's computer

On the author's computer, the command `pymanager list` (the manager's full name, see the WARNING box below) gave this on 2026-10-10. The text below is **copied from the real output**, not generated when the book was built:

```
Tag      Name                 Managed By  Version  Alias
-- No runtimes. Use 'py install <version>' to install one. --

* These runtimes were found, but cannot be updated or uninstalled. *
3.14  *  Python 3.14 (64-bit)  PythonCore  3.14.8
3.12     Python 3.12 (64-bit)  PythonCore  3.12.10
```

Meaning: the manager itself has not installed any Python yet, but it **found** two Pythons installed the old way (3.14.8 and 3.12.10). The asterisk `*` marks the default. Both can still be used through `py`.

> ⚠ WARNING
>
> On the author's computer there are **three** files named `py.exe` at the same time: one in `C:\WINDOWS`, one in the old Python folder, one from the install manager. When the author typed `py list`, the **old** `py.exe` answered, which took `list` for a script name and replied *"can't open file … list"*. `pymanager list`, on the other hand, always calls the new manager. The official documentation notes the same symptom: the legacy launcher can take priority, and the cure is to uninstall "Python launcher" from **Installed apps**. If your `py install` or `py list` is not recognised, use `pymanager` instead.

Checking the version

```
py --version
```

On the author's computer: `Python 3.14.8`. This works with both the old and the new launcher. The manager's documentation does not discuss `py --version` specifically; the proof comes only from the author's test above.

The PATH folders

The documentation lists adding `%LocalAppData%\Python\bin` to the **PATH** (the list of folders where Windows looks for programs) as optional, not required. Also keep `%UserProfile%\AppData\Local\Microsoft\WindowsApps` on the PATH, because removing it can break shortcuts.

Installing from a script (optional)

For automated installation, the official documentation suggests WinGet:

```
winget install 9NQ7512CXL7T -e --accept-package-agreements --disable-interactivity
py install --configure -y
```

This code was **not run** when the book was built because it changes the author's computer, so no output is shown; the commands were copied from the official documentation above.

Linux

Status: not tested on Linux. The content of this section comes from the official documentation (docs.python.org/3.14/using/unix.html, read 2026-10-10).

Python from distribution packages

The official documentation states that Python comes preinstalled on most Linux distributions and is available as a package on the others. It does **not** give an install command for Python itself (it leaves that to each distribution's guides). First check what you have:

```
$ python3 --version
```

The version may not be 3.14: that depends on your distribution and release, and I cannot confirm it for your distro. This book was run with Python 3.14.8; nearly all scripts were also tried on 3.12.10 with the same results (see the TIP box in Chapter 4), but Chapter 47 covers features that exist only in certain versions.

The official documentation does give commands for **IDLE** (Python's built-in editor), which is sometimes not installed along with Python:

```
$ sudo apt update && sudo apt install idle          # Debian and Ubuntu
$ sudo dnf install python3-idle                     # Fedora, RHEL, CentOS
$ sudo zypper install python3-idle                  # SUSE and openSUSE
$ sudo apk add python3-idle                         # Alpine
```

> **i** REMEMBER
>
> On Linux, `python3 -m venv` and `pip` are sometimes provided as **separate** packages by the distribution. The official Python packaging guide writes: *"On Linux, you may have to install an additional package such as `python3-pip`"*. The exact package name for venv on your distro (on Debian and Ubuntu probably `python3-venv`) is **not verified against distribution documentation**; if `python3 -m venv` complains about `ensurepip`, look for your distribution's venv package.

pyenv: installing many Python versions

pyenv installs several Python versions side by side by compiling them. According to github.com/pyenv/pyenv (read 2026-10-10):

```
$ curl -fsSL https://pyenv.run | bash
$ pyenv install -l          # list versions that can be installed
$ pyenv install 3.14        # a version prefix maps to the latest matching release
$ pyenv global 3.14
```

Before `pyenv install`, **install the build dependencies first**; the list is in the pyenv wiki, section "Suggested build environment". Pyenv also needs shell setup (the line `eval "$(pyenv init - bash)"` in `~/.bashrc`, or the equivalent for zsh and fish; see its README). Because pyenv compiles Python, installation can take several minutes.

Note: the pyenv README states that pyenv **does not support Windows** outside WSL (Windows Subsystem for Linux); for Windows it recommends a fork called `pyenv-win`. This book does not cover `pyenv-win`.

Building from source

The official documentation gives the sequence `./configure`, `make`, `make install`, but **recommends `make altinstall`** so that the system's own `python3` file is not overwritten. This is an advanced route; beginners should use `uv` or pyenv.

macOS

Status: not tested on macOS. Sources: docs.python.org/3.14/using/mac.html and docs.brew.sh/Language-Runtimes-and-Packages, read 2026-10-10.

The installer from python.org

Download the installer from python.org/downloads and double-click it. According to the official documentation, this installer is *universal2* (it runs on both Apple Silicon and Intel Macs) and needs a macOS account with **Administrator** rights. The result: a `Python 3.14` folder in *Applications* (containing IDLE and Python Launcher), a framework in `/Library/Frameworks/Python.framework`, and links in `/usr/local/bin/`.

Afterwards, **double-click `Install Certificates.command`** in the folder `/Applications/Python 3.14/`. This installs security certificates so Python can download from the internet. The sign of success: the text `Successfully installed certifi` and `update complete` appears.

Run Python with `python3` (or `python3.14`):

```
$ python3 --version
```

> ⚠ WARNING
>
> macOS ships Apple's own `/usr/bin/python3`. The official documentation calls it "usually older and incomplete", made for Apple's developer tools, and says it must **not be modified or deleted**. After installing from python.org, your `python3` should already point to the new one; if in doubt, check with `which python3`.

Homebrew

Homebrew is a package manager for macOS (brew.sh). The Python documentation lists it as an alternative distribution that is not maintained by the core Python team. The Homebrew documentation says its default Python formula installs `python3` and `pip3`, and advises installing **project dependencies in a venv**, not in Homebrew's base environment (which is marked "externally managed" under PEP 668). The Homebrew documentation I read does **not** literally contain the command `brew install python` (it tells you to use `brew info python` to see the current version); the VS Code guide writes `brew install python3`. So check with `brew info python` first.

uv: the shortcut on every system

uv (introduced in Chapter 17) can install Python and also manage projects. According to docs.astral.sh/uv (read 2026-10-10), to install uv:

```
# macOS and Linux
$ curl -LsSf https://astral.sh/uv/install.sh | sh

# Windows (PowerShell)
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"

# alternatives: Homebrew (macOS) or WinGet (Windows)
$ brew install uv
winget install --id=astral-sh.uv -e
```

Then Python itself (from the uv guide):

```
uv python install 3.14      # install Python 3.14
uv python list              # see what is available and installed
uv venv                     # create a venv; installs Python first if missing
```

According to that guide, uv uses a Python already on the system when it fits and, by default, downloads the Python it needs. On the author's computer, `uv --version` answers `uv 0.11.28`; uv was already installed, so installing uv itself was **not retested**.

TECHNICAL STUFF (feel free to skip)

The command `curl ... | sh` downloads a script and runs it immediately. It is official and common, but you should get used to **reading the script first** (open its address in a browser) or using a package manager (`brew`, `winget`). Try it first on a computer that is not a production machine.

Virtual environments (venv) on each system

Chapter 17 explains *why* every project needs a venv. This is a summary of *how* on each system, from the official packaging guide and docs.python.org/3.14/library/venv.html (read 2026-10-10):

Step	Windows (PowerShell)	Linux and macOS (bash/zsh)
Create	`py -m venv .venv`	`python3 -m venv .venv`
Activate	`.venv\Scripts\Activate.ps1`	`source .venv/bin/activate`
In `cmd.exe`	`.venv\Scripts\activate.bat`	(not applicable)
Check	`where python`	`which python`
Check pip	`py -m pip --version`	`python3 -m pip --version`
Deactivate	`deactivate`	`deactivate`

Once active, the venv folder name appears at the start of the prompt, and `python` points to the Python inside the venv.

TIP

You do **not have to activate** a venv. The official documentation says you can simply call the Python inside it by full path, for example `.venv\Scripts\python.exe script.py` on Windows or `.venv/bin/python script.py` on Linux and macOS. That is how this book's project is run.

A real test: creating a venv from a script

Python can create a venv through the `venv` module (the same one the `-m venv` command uses). This script builds one in the `_tmp/` folder, then inspects its contents:

```
# Appendix A - Creating a virtual environment (venv) from Python
import shutil
import subprocess
import sys
import venv
from pathlib import Path

folder = Path(__file__).parent / "_tmp" / "venv_example"
if folder.exists():
    shutil.rmtree(folder)  # only this script's own example folder

venv.create(folder, with_pip=True)
print("Environment created:", folder.exists())

# On Windows the programs live in Scripts/, on Linux and macOS in bin/
sub = "Scripts" if sys.platform == "win32" else "bin"
contents = sorted(p.name for p in (folder / sub).iterdir())
print("Contents of folder", sub + ":", ", ".join(contents[:6]), "...")

venv_python = folder / sub / ("python.exe" if sys.platform == "win32" else "python")
for arguments in (["--version"], ["-m", "pip", "--version"]):
```

```
      result = subprocess.run([str(venv_python), *arguments], capture_output=True,
text=True)
    text = result.stdout.strip()
    label = "python " + " ".join(arguments[:2])
    print(f"{label:<18}->", text.split(" from ")[0])  # drop the path
```

```
Environment created: True
Contents of folder Scripts: Activate.ps1, activate, activate.bat, activate.fish,
deactivate.bat, pip.exe ...
python --version   -> Python 3.14.8
python -m pip      -> pip 26.2.1
```

The first line proves the venv was created. The second shows the contents of the `Scripts` folder on Windows: there are `activate`, `Activate.ps1`, `activate.bat`, and so on (on Linux and macOS the activation scripts are in `bin/`). The last two lines call Python and pip **inside the venv** directly, without activation. In this fresh venv pip was installed along with it. The pip version number will change over time; what matters is that it answers.

> ⚠ WARNING
>
> In Windows PowerShell, `Activate.ps1` is sometimes refused with a message about the *execution policy* (the rule for running scripts). The official documentation names the cure: `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser`. That **changes a security setting of your account**, so consider other routes: use `cmd.exe` with `activate.bat`, or do not activate the venv at all (see the TIP above). The policy on the author's computer was not tested for this case.

Checking your environment

Run this checker script on your computer. It records the Python version, the operating system, whether needed modules are present, and whether some commands are found on the PATH:

```
# Appendix A - Checking the Python environment on your computer
import platform
import shutil
import sys

print(f"{'Python version':<16}: {platform.python_version()}")
print(f"{'Operating system':<16}: {platform.system()}")
print(f"{'Architecture':<16}: {platform.machine()}")
print(f"{'Series 3.14?':<16}: {'yes' if sys.version_info[:2] == (3, 14) else 'no'}")

# Standard modules needed for virtual environments
for module in ("venv", "ensurepip", "tkinter", "sqlite3"):
    try:
        __import__(module)
        status = "present"
    except ImportError:
        status = "MISSING"
    print(f"{'module ' + module:<16}: {status}")

# Commands searched for on PATH (the result differs on every computer)
for command in ("python", "py", "uv", "git", "code"):
    status = "found" if shutil.which(command) else "not found"
    print(f"{'command ' + command:<16}: {status}")
```

```
Python version  : 3.14.8
Operating system: Windows
Architecture    : AMD64
```

```
Series 3.14?    : yes
module venv     : present
module ensurepip: present
module tkinter  : present
module sqlite3  : present
command python  : found
command py      : found
command uv      : found
command git     : found
command code    : found
```

This is the output of **the author's computer**; yours will differ on the operating system, architecture, and found-command lines. How to read it:

- **Python version** and **Series 3.14?** tell you whether you are on the book's reference version.
- **module venv / ensurepip** must be `present`. If `MISSING` on Linux, see the REMEMBER box in the Linux section.
- **module tkinter** is only needed for the windowed interface example in Chapter 42; **sqlite3** for Chapter 24.
- **command …** `found` means it is on the PATH. `not found` is not an error: it only means that command is not installed yet or not on the PATH.

Other tools this book uses

Tool	What for	How to install	Required?
VS Code	Code editor	code.visualstudio.com	No (Notepad is enough for beginners)
Git	Version tracker (Chapter 20)	git-scm.com/downloads	From Chapter 20
Quarto	Building this book	-	**Not needed** to read or follow the book

The VS Code guide (read 2026-10-10) says: install the Python extension, then choose the interpreter through the Command Palette with the command **Python: Select Interpreter**. For Windows it checks the installation with `py -3 --version`. On the author's computer VS Code and Git are installed (`code` and `git` are found by the checker script; `git --version` answers `git version 2.55.0.windows.3`), but the process of **installing them** was not retested.

Heavy tools needed only by particular chapters are explained in those chapters: PostgreSQL and PostGIS (Chapters 24 and 40), QGIS (Chapter 40), Redis (Chapter 67), Docker (Chapter 48), and the NVIDIA driver and CUDA (Chapter 55). Install them **only when you reach that chapter**.

Common problems and their solutions

Symptom	Likely cause	What to try
`py` not recognised (Windows)	Install manager not installed, or alias turned off	Install the manager (see the Windows section); open a **new** terminal after installing
`py list` / `py install` answers "can't open file"	The old `py.exe` takes priority	Use `pymanager list`; or uninstall the old "Python launcher" from Installed apps (per the official documentation)
`python` opens the Microsoft Store (Windows)	Store alias active without Python installed	Install Python first; the `py` command can still be used

Symptom	Likely cause	What to try
`python` not found (Linux/macOS)	Only `python3` exists	Type `python3`; inside an active venv, `python` is available
`pip` gives `externally-managed-environment`	Distro/Homebrew Python forbids global installs (PEP 668)	**Create a venv**, then install there; do not use `--break-system-packages`
`python3 -m venv` fails about `ensurepip` (Linux)	The distro's venv package is not installed	Look for your distro's venv package (package name not verified)
`Activate.ps1` refused	PowerShell execution policy	Use `activate.bat` in `cmd`, or call the venv's `python.exe` directly
Library fails to install because there is no compiler	No ready-made package (*wheel*) for your Python/OS	Try an older Python, `uv`, or conda (Chapters 17 and 38)
SSL certificate fails (macOS python.org)	Forgot to run `Install Certificates.command`	Run that file (see the macOS section)
Script "cannot find module" although it is installed	Installed in a different venv/Python than the one running the script	Run `python -m pip install ...` with the same Python; check `which python` / `where python`
Venv broken after the folder was moved	Venvs cannot be moved or copied	Recreate the venv (per the official documentation) and reinstall from `requirements.txt`

The rows about `py list`, `externally-managed-environment`, `Activate.ps1`, a moved venv, and SSL certificates come from the official documentation quoted in this appendix. The other rows are common experience that I did not test on the author's computer (label: **probable**); the venv package name on Linux is even **not verified**.

> **i** REMEMBER
>
> When confused, **go back to basics**: (1) `py --version` or `python3 --version`, (2) create a venv in the project folder, (3) call the Python inside the venv by full path. Those three steps solve most installation problems.

Next: Appendix B summarises the key commands in cheat sheets. If you are looking for a study order, jump to Appendix E.

Learn Python from zero, step by step

- Computers, the terminal, and how a program thinks
- Variables, conditionals, loops, and functions
- Files, data formats, and office document automation
- Examples from forests, orchards, rice fields, ponds, and barns

---

Volume 1 takes you from an empty computer to writing small programs that really run. Every chapter uses made-up data and exercises you can run right away.

Compiled by

Badar Mubarok Yogaswara