Boring Python

K19G

2026-09-08

Table of contents

Boring Python

A linear guide to writing Python the way it lasts in real projects: **simple, explicit, and a little boring**. Clarity beats novelty. Readability beats magic. The boring default is the one a teammate can run, test, and change six months later.

Baseline: Python **3.14** · toolchain `uv` · format/lint `ruff` · tests `pytest`. Completely self-contained. You do not need any other title in this library.

Who this is for

- People who have never written Python and want a complete path, not a bag of tricks.
- Working developers who can already `print("hi")` but want better defaults.
- Anyone building scripts, CLIs, or services who is tired of clever one-liners and hidden frameworks.

You do not need prior Python. You do need a terminal, a text editor, and a willingness to type the examples.

What "boring" means here

Python is a large language with a huge standard library. Well-written Python still looks ordinary: names are clear, functions are short, exceptions are specific, types appear where they help, and async is used when you have I/O to wait on — not because it looks modern.

Each chapter teaches **the default that lasts**, then shows the trap that looks smarter for a week.

This book is the language, the toolchain, the standard library you actually use, tests, concurrency, and shipping. It is not a tour of web frameworks, dataframes, or ML stacks.

How to read

1. Read a chapter.
2. Copy the program into the filename in the comment.
3. Run it with `uv run python that_file.py` (or `uv run pytest` when the chapter says so).
4. Change one thing. Run it again.
5. Do the **Try this** exercises before moving on.

The book is designed to be read **front to back**. You can jump to exceptions, typing, or asyncio if you already write Python — but the early chapters define the vocabulary later chapters assume.

How examples work

Every Python listing the reader is told to run is a **complete, runnable program** (or a complete test file). Nothing is a fragment that "you should imagine the rest of."

```python
# hello.py
def main():
    print("hello, desk")


if __name__ == "__main__":
    main()
```

```
uv run python hello.py
```

```
hello, desk
```

When a chapter needs more than one file (`pyproject.toml` plus `main.py`, or a test beside the code), both files are shown **in the chapter**. There is no separate examples directory.

A small **desk** (orders, tickets, shifts) shows up across chapters so types, classes, exceptions, and tests feel like one codebase — but each listing still runs on its own.

Early chapters skip type annotations. Types arrive as their own part, then stay.

Map of the book

Part	Folder	What you leave with
1	`01-foundations`	Why Python exists, install with `uv`, modules, a language tour
2	`02-toolchain`	`uv`, `ruff`, `pyproject.toml`, environments
3	`03-types-and-values`	Built-in types, `None`, literals, scope
4	`04-control-flow`	`if`, `match`, loops, comprehensions
5	`05-collections`	lists, tuples, strings, dicts, sets, dataclasses
6	`06-functions`	signatures, first-class functions, decorators
7	`07-modules`	imports, packages, `__main__`
8	`08-objects`	classes, composition, properties, the data model
9	`09-typing`	hints, protocols, generics
10	`10-errors`	exceptions, groups, context managers
11	`11-iteration`	iterators, generators, itertools
12	`12-testing`	pytest, fixtures, mocks, integration
13	`13-concurrency`	threads, asyncio, processes, guidelines
14	`14-stdlib`	pathlib, I/O, datetime, JSON, HTTP, logging, subprocess
15	`15-shipping`	ruff/audit, packaging, release, docs

Part	Folder	What you leave with
16	`16-long-term`	inspect, FFI, profiling, design that lasts
99	`99-appendices`	glossary and command cheat sheet

What this book is not

- Not a framework tutorial.
- Not a catalog of every standard-library module.
- Not a collection of interview puzzles.
- Not a CPython-internals encyclopedia.

It is a guide to writing Python that stays understandable when the requirements change.

Formats

HTML, PDF, and EPUB via the library portal.

Foundations

Why Python Exists

Why Python Exists

Python exists because people wanted a language that reads like notes you would leave a colleague: **obvious names, as little ceremony as possible, batteries included**. The boring default in this book is the same default Python was built for: code a stranger can run on Monday morning.

Mental model

Python is an interpreted, dynamically typed language with garbage collection and a large standard library. You trade a compile step and static types-by-default for:

- a short path from idea to running program
- one obvious way to do many everyday jobs (files, JSON, HTTP, processes)
- a single runtime you can inspect (`type`, `dir`, `help`)
- exceptions you can see in the traceback

That trade is the point. If you keep reaching for magic metaclass frameworks on day one, you will dislike this book. If you like software that stays boring as it grows, you will like it.

A **terminal** is the window where you type commands. `PATH` is the list of directories the shell searches for a program named `uv` or `python`. A **file path** is a location on disk (`hello.py`, `/home/you/desk`).

Worked examples

Case 1: A complete program, no ceremony

Save as `hello.py`. This is the smallest useful Python program: a function, a print, and the usual `main` guard.

```
# hello.py
def main():
    print("desk is open")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python hello.py
```

Output:

```
desk is open
```

`uv run` picks a Python (3.14 in this book) and runs the file. There is no compile-to-binary step you have to wait on. What you ran is the source.

The `if __name__ == "__main__"` line means "only call `main` when this file is the program," not when another file imports it. Keep that guard. It is boring and it prevents accidents.

Case 2: Explicit failure, visible traceback

Python uses **exceptions** for failure. This program opens tables by number. Zero is nonsense, so it raises. The traceback names the file, the line, and the exception type.

```
# open_tables.py
def open_table(n):
    if n <= 0:
        raise ValueError(f"table {n}: number must be positive")
    print(f"opened table {n}")


def main():
    tables = [3, 0, 11]
    for n in tables:
        open_table(n)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python open_tables.py
```

Output (the process exits non-zero):

```
opened table 3
Traceback (most recent call last):
  File "open_tables.py", line 16, in <module>
    main()
  File "open_tables.py", line 12, in main
    open_table(n)
  File "open_tables.py", line 4, in open_table
    raise ValueError(f"table {n}: number must be positive")
```

```
ValueError: table 0: number must be positive
```

The line numbers in your traceback follow the file you saved. Read it from the bottom: the exception, then the call that caused it.

Case 3: Data that looks like the domain

Python's built-in types are enough for a lot of desk work. This program models a ticket as a dict, then prints a label. Later chapters will replace the dict with a dataclass. Start here.

```
# ticket.py
def label(ticket):
    return f"ticket {ticket['id']} → table {ticket['table']}"


def main():
    t = {"id": 7, "table": 12}
    print(label(t))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket.py
```

Output:

```
ticket 7 → table 12
```

Case 4: What Python will let you do that you should not

This program works. It is also a waste: a class, a constructor, and a method whose only job is `print`.

```
# too_clever.py
class Printer:
    def __init__(self, stream):
        self.stream = stream

    def emit(self, message):
        print(message, file=self.stream)


def main():
```

```
    import sys

    Printer(sys.stdout).emit("desk is open")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python too_clever.py
```

Output:

```
desk is open
```

The boring version is Case 1. Introduce a class when you have **state that lives** and **behavior that belongs to it** — not when you have a slogan about objects.

The trap

Coming from a language that prizes types or from a tutorial that starts with a web framework, it is tempting to wrap every script in a package, a plugin system, and a settings object on day one. That is how a 20-line desk tool becomes a 12-module "platform."

Write the function. Run it. Add structure when a name collides or a test needs a seam.

The boring rule

- Prefer a function and a dict or dataclass over a hierarchy.
- Raise a specific exception (`ValueError`, `TypeError`) for the caller's mistake. Do not return `"error"` strings as the main API.
- Keep the `main` guard.
- Split files when names collide or when a boundary is real.
- If the standard library already does the job, do not add a dependency yet.

Try this

1. Change `hello.py` so it prints the number of tables (an `int`) next to the message. Use an f-string.
2. In `open_tables.py`, catch `ValueError` around `open_table`, print the error, and continue to the next table.
3. In `ticket.py`, add a `status` key (`"open"` / `"paid"`) and print it in `label`.

Installing and Running Python

Installing and Running Python

The boring way to run Python in this book is `uv`: install it once from astral.sh, pin **Python 3.14**, and run every file with `uv run python`. You do not need a distro package named `python`, a GUI installer, or an IDE before the first script works.

Mental model

A **runtime** is the program that reads a `.py` file and executes it. `uv` is one tool that:

- installs that runtime (`uv python install 3.14`)
- writes down the version the folder expects (`.python-version`, `pyproject.toml`)
- runs a file with that interpreter (`uv run python shift.py`)

A **terminal** is the window where you type commands. `PATH` is the list of directories the shell searches for a program named `uv`. On macOS and Linux the installer puts `uv` in `~/.local/bin`. If the shell says `uv: command not found` right after install, close the terminal and open a new one so `PATH` reloads.

Python source is **text**. An **editor** is anything that saves a `.py` file. Use the editor you already use.

Worked examples

Case 1: Install `uv`, then Python 3.14

These are terminal commands, not Python programs.

macOS and Linux:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows (PowerShell):

```
irm https://astral.sh/uv/install.ps1 | iex
```

Confirm `uv` is on `PATH`, then install the interpreter this book uses:

```
uv --version
uv python install 3.14
```

```
uv python list
```

You should see a 3.14 row. In a project folder, pin it so `uv run` keeps choosing 3.14:

```
uv python pin 3.14
```

That writes a tiny file named `.python-version` whose whole contents are:

```
3.14
```

Case 2: Prove the runtime

Save as `runtime.py`.

```
# runtime.py
import sys


def main():
    major, minor = sys.version_info[:2]
    print(f"python {major}.{minor}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python runtime.py
```

Output:

```
python 3.14
```

`uv run` selects a 3.14 interpreter (and will download one if needed), then runs the file. You did not activate a virtual environment by hand.

Case 3: A first desk script

Save as `shift.py`. Open it in your editor — Zed, VS Code, Neovim, Notepad, TextEdit, whatever you already have. If it can save this file, it is enough.

```
# shift.py
def main():
    name = "nights"
    tables = 8
```

```
    print(f"shift {name}: {tables} tables")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift.py
```

Output:

```
shift nights: 8 tables
```

Change `tables` to `9`. Save. Run the same command. The number in the output changes. Edit, save, run: that loop is the whole workflow.

Case 4: Write the project down

A project is a folder with a **`pyproject.toml`**. You can write the file yourself, or let `uv` create the same shape:

```
uv init --bare --name desk --python 3.14
```

Save this next to `shift.py` as `pyproject.toml`:

```
# pyproject.toml
[project]
name = "desk"
version = "0.1.0"
requires-python = ">=3.14"
dependencies = []
```

Then:

```
uv python pin 3.14
uv run python shift.py
```

Same output as Case 3. `requires-python` is how you tell a teammate which Python this folder expects. `dependencies` stays empty until a later chapter needs a package.

A typical first folder looks like this:

```
desk/
  .python-version
  pyproject.toml
  shift.py
```

That is enough. Do not add `src/`, a package name, or a global "system Python" yet.

Case 5: Another file, same command

Save as `ticket.py`.

```
# ticket.py
def main():
    ticket_id = 41
    table = 6
    print(f"ticket {ticket_id} sits at table {table}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket.py
```

Output:

```
ticket 41 sits at table 6
```

Every example in this book that you are told to run uses that same command shape: `uv run python` and the filename.

The trap

Installing "Python" from a website, then another copy from a package manager, then running `python3 shift.py` with whichever binary is first on `PATH`. Six months later one machine is 3.11, another is 3.14, and a forgotten `pip install` wrote packages into a user directory you cannot name.

A second trap: delaying the first script until you have chosen an IDE, a theme, and a plugin pack. The language is text. Highlighting can wait.

The boring rule

- Install **`uv` from https://astral.sh/uv** with the script above.
- Install the interpreter with `uv python install 3.14`.
- Pin it in the project with `uv python pin 3.14`.
- Keep a `pyproject.toml` with `requires-python = ">=3.14"`.
- Run every example with `uv run python file.py`.
- Use the editor you already use. Save as `.py`.

- If `uv` is not found, open a new terminal so `PATH` updates.

Try this

1. In `runtime.py`, also print `sys.executable`. Run it and read the path — that is the interpreter `uv` chose.
2. In `shift.py`, add a second print for the closer's name (a string).
3. Run `uv run python --version` in the same folder as `pyproject.toml`. Confirm it reports 3.14.

How Python Code Is Organized

How Python Code Is Organized

Python code lives in **files**. You run a file, or you import it. The boring default is a **flat folder**: a few `.py` files next to `pyproject.toml`. Do not start with a `src/` tree or a package named after your company.

Mental model

Three words, used precisely:

- A **script** is a file you run (`uv run python main.py`).
- A **module** is a file you import (`import greet`). Every `.py` file is a module.
- A **package** is a directory of modules (usually with an `__init__.py`). You need one when a single folder of files is no longer a clear boundary.

When Python loads a file, it sets a string named `__name__`. If that file is the program, `__name__` is `"__main__"`. If another file imported it, `__name__` is the module name (`"greet"`). The usual guard means "run `main` only when this file is the program."

Flat layout (boring):

```
desk/
  pyproject.toml
  greet.py
  main.py
```

`src/` layout (later, when you *install* the package):

```
desk/
  pyproject.toml
  src/
    desk/
      __init__.py
      greet.py
```

Stay flat until a name collides or you are shipping an installable library.

Worked examples

Case 1: One file is a script and a module

Save as `greet.py`. It defines a function. It also has a `main` guard so you can run it by itself.

```
# greet.py
def desk_hello(name):
    return f"hello, {name}"


def main():
    print(desk_hello("desk"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python greet.py
```

Output:

```
hello, desk
```

Case 2: A second file imports the first

Save as `main.py` **in the same folder** as `greet.py`.

```
# main.py
import greet


def main():
    print(greet.desk_hello("nights"))
    print("greet.__name__ =", greet.__name__)
    print("this file __name__ =", __name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python main.py
```

Output:

```
hello, nights
greet.__name__ = greet
this file __name__ = __main__
```

`import greet` loads `greet.py` once, binds the module to the name `greet`, and does **not** call `greet.main`, because inside `greet.py` the name is `"greet"`, not `"__main__"`. That is why the guard exists.

The same folder still has `pyproject.toml`:

```toml
# pyproject.toml
[project]
name = "desk"
version = "0.1.0"
requires-python = ">=3.14"
dependencies = []
```

Python finds `greet` because the directory of the script you ran is on the import path. You did not need `src/`.

Case 3: `__name__` is a string you can print

Save as `whoami.py`.

```python
# whoami.py
def main():
    print(__name__)


if __name__ == "__main__":
    main()
```

Run as a script:

```
uv run python whoami.py
```

Output:

```
__main__
```

Load it as a module with `-m` (the next part covers `-m` more fully). From the same folder:

```
uv run python -c "import whoami; print(whoami.__name__)"
```

Output:

```
whoami
```

Same file, two jobs. Keep the guard so those jobs stay distinct.

Case 4: A package is a directory, not a personality

When you eventually need a package, it is a folder of modules. The boring empty `__init__.py` is enough:

```
desk/
  pyproject.toml
  desk/
    __init__.py
    greet.py
```

`desk/__init__.py` can be empty. `import desk.greet` then works after the package is on the path. You do **not** need this for the examples in this chapter. Put `greet.py` next to `main.py` until a third file makes the boundary obvious.

The trap

Work at import time. Save as `noisy.py`:

```python
# noisy.py
print("loading noisy")


def ping():
    return "pong"
```

Save as `use_noisy.py`:

```python
# use_noisy.py
import noisy


def main():
    print(noisy.ping())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python use_noisy.py
```

Output:

```
loading noisy
pong
```

The print in `noisy.py` ran because import **executes the file**. A second import of `noisy` in the same process would not print again (modules are cached), which makes the surprise even harder to see. Put work in functions. Call those functions from `main`.

The other common trap is the opposite: a `src/desk/core/utils/helpers` tree on day one, with one function in it. Names collide later, not on line one. Split files when you have a reason.

The boring rule

- One folder, several `.py` files, a `pyproject.toml`. Flat until you need a package.
- `import name` for a sibling module. Call `name.function`.
- Keep `if __name__ == "__main__":` on every file you run.
- Do not do real work (prints, network, files) at module top level.
- Do not use `from greet import *`.
- Do not start a `src/` layout until you are packaging something you install.

Try this

1. In `greet.py`, add `desk_bye(name)` that returns `f"bye, {name}"`. Call it from `main.py`.
2. Run `uv run python greet.py` again after Case 2. Confirm it still prints `hello, desk` and does not print the lines from `main.py`.
3. Move the `print("loading noisy")` in `noisy.py` into `ping`. Run `use_noisy.py` and notice import is now silent.

Python Tour

Python Tour

This chapter is a walking tour: **names, `if`/`for`, lists and dicts, a function, an exception, f-strings**. It is not a complete language. Later parts slow down on each piece. No classes, no type hints, no async.

Mental model

A **name** refers to a value (`closer = "Sam"`). Values have types (`str`, `int`, `list`, `dict`) that you can ask for with `type`, but you do not annotate them yet.

Control flow is `if`/`else` to choose, and `for` to walk a collection.

A **list** is an ordered sequence. A **dict** is a map from key to value. Desk data starts here: a ticket is a dict, a shift's tickets are a list of dicts.

A **function** is a named piece of work with arguments and a `return`. An **exception** is how failure is reported. An **f-string** (`f"...{name}..."`) is how you build a message from values.

Worked examples

Case 1: Names and f-strings

Save as `hours.py`.

```
# hours.py
def main():
    closer = "Sam"
    start = 17
    end = 23
    length = end - start
    print(f"{closer} works {start}:00-{end}:00 ({length} hours)")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python hours.py
```

Output:

```
Sam works 17:00-23:00 (6 hours)
```

`closer` is a string. `start`, `end`, and `length` are integers. The f-string interpolates them in order.

Case 2: `if` and `for` over a list of dicts

Save as `tickets.py`.

```python
# tickets.py
def main():
    tickets = [
        {"id": 1, "status": "open"},
        {"id": 2, "status": "paid"},
        {"id": 3, "status": "open"},
    ]
    for t in tickets:
        if t["status"] == "open":
            print(f"ticket {t['id']} still open")
        else:
            print(f"ticket {t['id']} is {t['status']}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python tickets.py
```

Output:

```
ticket 1 still open
ticket 2 is paid
ticket 3 still open
```

`tickets` is a list. Each `t` is a dict. `t["id"]` looks up a key. `for` walks the list once. `if` branches on the status string.

Case 3: A function over order lines

Save as `orders.py`. The function does one job: quantity times price. `main` prints the bill.

```
# orders.py
def line_total(qty, price):
    return qty * price


def main():
    lines = [
        {"item": "soup", "qty": 2, "price": 6},
        {"item": "pie", "qty": 1, "price": 9},
    ]
    total = 0
    for line in lines:
        amount = line_total(line["qty"], line["price"])
        total = total + amount
        print(f"{line['qty']}× {line['item']} = {amount}")
    print(f"total {total}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python orders.py
```

Output:

```
2× soup = 12
1× pie = 9
total 21
```

`return` sends a value back to the caller. `total = total + amount` is a running sum. A later chapter will show `+=` and dataclasses. This is enough to bill a table.

Case 4: Raise when the input is nonsense

Save as `bad_price.py`.

```
# bad_price.py
def line_total(qty, price):
    if price < 0:
        raise ValueError(f"price {price} cannot be negative")
    return qty * price


def main():
```

```
    print(line_total(2, 6))
    print(line_total(1, -9))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python bad_price.py
```

Output (the process exits non-zero):

```
12
Traceback (most recent call last):
  File "bad_price.py", line 14, in <module>
    main()
    ~~~~^^
  File "bad_price.py", line 10, in main
    print(line_total(1, -9))
          ~~~~~~~~~~^^^^^^^
  File "bad_price.py", line 4, in line_total
    raise ValueError(f"price {price} cannot be negative")
ValueError: price -9 cannot be negative
```

Your editor may wrap the caret lines. Read the traceback from the **bottom**: the exception type and message, then the call that caused it. `ValueError` is the right type when the caller passed a bad value.

The trap

Cramming the tour into one clever line: a lambda, a side-effecting list comprehension, and a `print` inside an expression. It fits on a slide. It does not fit a desk.

The other trap is reaching for a class the moment you have two dicts. `{"id": 1, "status": "open"}` is fine. A class arrives when you have behavior that belongs with the data, not before.

The boring rule

- Names are ordinary words (`closer`, `tickets`, `line_total`).
- Use f-strings for messages. Do not stitch with `+` unless you are joining a list.
- A list of dicts is a valid model for tickets and order lines.
- Functions `return` values. They do not print unless printing *is* the job (`main`).
- Raise `ValueError` (or another specific type) for a bad argument. Do not return `"error"`.
- Keep `if __name__ == "__main__":`.

Try this

1. In `hours.py`, add a `break_minutes` integer and print it in the same f-string.
2. In `tickets.py`, count how many tickets are `"open"` and print the count after the loop.
3. In `orders.py`, add a third line (`"tea"`, qty `3`, price `2`) and confirm the total becomes `27`.
4. In `bad_price.py`, wrap the second `line_total` call in `try/except ValueError` and print the error without crashing.

Toolchain

Core Commands

Core Commands

The toolchain in this book is small on purpose: **`uv run`**, **`uv python`**, **`python -c`**, **`python -m`**, **`ruff`**, **`pytest`**. Learn these six and you can run, format, lint, and test every later chapter.

Mental model

`uv` is the front door. `uv run …` means "in this project, with this Python, run that command." You rarely type a bare `python`.

- `uv run python file.py` — run a script.
- `uv run python -c "…"` — run one statement, no file.
- `uv run python -m name` — run a *module* by name (`json.tool`, or `open_shift` for `open_shift.py`).
- `uv python install|pin|list` — manage interpreters.
- `uv run ruff format|check` — format and lint.
- `uv run pytest` — run tests.

`ruff` and `pytest` are ordinary packages. Add them once as dev dependencies, then `uv run` finds them.

Worked examples

Case 1: `uv run` and `uv python`

Save as `open_shift.py`.

```
# open_shift.py
def main():
    print("shift open")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python open_shift.py
```

Output:

```
shift open
```

Useful `uv python` commands (output varies by machine):

```
uv python install 3.14
uv python pin 3.14
uv python list
uv run python --version
```

The last one should report `Python 3.14.x`. That is the check that matters.

Case 2: `python -c` (no file)

One-off prints and arithmetic do not need a file.

```
uv run python -c 'print("shift open")'
```

Output:

```
shift open
```

Keep `-c` for probes. Put anything you will run twice in a `.py` file.

Case 3: `python -m` runs a module by name

The same `open_shift.py` can be run as a module. The name is the filename without `.py`.

```
uv run python -m open_shift
```

Output:

```
shift open
```

`-m` is also how you run tools that shipped as modules. Save as `ticket_dump.py`:

```
# ticket_dump.py
import json


def main():
    print(json.dumps({"id": 7, "table": 12}))


if __name__ == "__main__":
    main()
```

Pretty-print its JSON with the standard library's `json.tool`:

```
uv run python ticket_dump.py | uv run python -m json.tool
```

Output:

```
{
    "id": 7,
    "table": 12
}
```

Case 4: `ruff format` and `ruff check`

Add the linter once (this also creates or updates `pyproject.toml` / `uv.lock`):

```
uv add --dev ruff
```

Save as `messy.py` — working code, ugly spacing:

```
# messy.py
def main( ):
    tables=8
    print( f"tables {tables}" )


if __name__=="__main__":
    main()
```

Format it in place:

```
uv run ruff format messy.py
```

Output:

```
1 file reformatted
```

`messy.py` on disk is now:

```
# messy.py
def main():
    tables = 8
    print(f"tables {tables}")


if __name__ == "__main__":
    main()
```

Save as `unused.py` for a lint finding:

```
# unused.py
import json


def main():
    tables = 8
    print(tables)


if __name__ == "__main__":
    main()
```

```
uv run ruff check unused.py
```

Output:

```
F401 [*] `json` imported but unused
 --> unused.py:2:8
  |
1 | # unused.py
2 | import json
  |        ^^^^
help: Remove unused import: `json`

Found 1 error.
[*] 1 fixable with the `--fix` option.
```

`F401` is "imported but unused." Delete the import, or run `uv run ruff check --fix unused.py`. The next chapter treats format and lint as policy.

Case 5: `pytest` on a tiny test

Save as `total.py`:

```
# total.py
def line_total(qty, price):
    return qty * price


def main():
    print(line_total(3, 4))


if __name__ == "__main__":
```

```
    main()
```

Save as `test_total.py` in the **same folder**:

```
# test_total.py
from total import line_total


def test_line_total():
    assert line_total(3, 4) == 12
```

```
uv add --dev pytest
uv run pytest test_total.py -q
```

Output (the time on the last line will vary):

```
.                                                                        [100%]
1 passed in 0.01s
```

A test file is a complete program pytest loads. Functions named `test_*` are collected. `assert` is the check. You do not add `if __name__ == "__main__"` to the test file.

The trap

Installing `ruff` and `pytest` "globally" with a random `pip`, then wondering why `uv run` cannot see them. `uv run` uses **this project's** environment. Add tools with `uv add --dev`.

The other trap is living in `python -c` and never saving a file. `-c` has no traceback you can open, no test, and no format.

The boring rule

- Run scripts with `uv run python file.py`.
- Probe with `uv run python -c '…'` only.
- Use `uv run python -m name` for modules and stdlib tools.
- Pin 3.14 with `uv python pin 3.14`.
- Format and lint with `uv run ruff format` and `uv run ruff check`.
- Test with `uv run pytest`.
- Add `ruff` and `pytest` as **dev** dependencies, not as runtime dependencies of the desk.

Try this

1. Change `open_shift.py` to print the table count. Run it both as `uv run python open_shift.py` and as `uv run python -m open_shift`.
2. In `test_total.py`, add `test_zero_qty` that asserts `line_total(0, 4) == 0`. Run pytest again.
3. Run `uv run ruff check unused.py --fix`, then `uv run ruff check unused.py` and confirm it is clean.

Formatting, Linting, and Documentation

Formatting, Linting, and Documentation

`ruff format` is not optional. You do not argue about spaces around `=`. You run the formatter. **`ruff check`** catches unused names and a handful of real mistakes. A **docstring** is a `"""triple-quoted string"""` right under a `def`, and `help()` reads it.

Mental model

Three layers, in this order:

1. **Format** — how the file looks (`ruff format`). Mechanical. No taste.
2. **Lint** — cheap static checks (`ruff check`). Unused imports, unused variables, a few footguns.
3. **Docs** — a sentence for a function another person (or you, in six months) will call. Not a restatement of the name.

`help(fn)` prints the function's signature and docstring. It works on your functions and on the built-ins.

Add the tool once:

```
uv add --dev ruff
```

Worked examples

Case 1: Format is non-negotiable

Save as `messy.py`. It runs. It looks like a fight.

```
# messy.py
def main( ):
    tables=8
    print( f"tables {tables}" )


if __name__=="__main__":
    main()
```

Run:

```
uv run ruff format messy.py
uv run python messy.py
```

Formatter output:

```
1 file reformatted
```

Program output:

```
tables 8
```

The file on disk is now:

```
# messy.py
def main():
    tables = 8
    print(f"tables {tables}")


if __name__ == "__main__":
    main()
```

Do not hand-match this style. Run `ruff format`. In a project, `uv run ruff format .` formats every Python file.

Case 2: A lint finding, then the fix

Save as `lint_bug.py`. The unused import is the bug `ruff check` is for.

```
# lint_bug.py
import json


def main():
    print("shift open")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python lint_bug.py
uv run ruff check lint_bug.py
```

The program still prints:

```
shift open
```

The linter reports:

```
F401 [*] `json` imported but unused
 --> lint_bug.py:2:8
  |
1 | # lint_bug.py
2 | import json
  |        ^^^^
help: Remove unused import: `json`

Found 1 error.
[*] 1 fixable with the `--fix` option.
```

Fix — save this as `lint_bug.py` (or run `uv run ruff check --fix lint_bug.py`):

```
# lint_bug.py
def main():
    print("shift open")


if __name__ == "__main__":
    main()
```

```
uv run ruff check lint_bug.py
uv run python lint_bug.py
```

Linter:

```
All checks passed!
```

Program:

```
shift open
```

`F401` is not a taste note. An unused import is a lie about what the file needs. Delete it.

Case 3: A docstring `help()` can read

Save as `ticket_help.py`. The first statement in the function is a string. That string is the docstring.

```python
# ticket_help.py
def label(ticket_id, table):
    """Return a one-line label for a ticket at a table."""
    return f"ticket {ticket_id} → table {table}"


def main():
    help(label)
    print(label(7, 12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_help.py
```

Output:

```
Help on function label in module __main__:

label(ticket_id, table)
    Return a one-line label for a ticket at a table.

ticket 7 → table 12
```

`help(label)` reads `label.__doc__`. You did not write a website. You wrote the sentence that shows up when someone asks the runtime.

Case 4: `help()` on a built-in

Save as `help_len.py`.

```python
# help_len.py
def main():
    help(len)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python help_len.py
```

Output (first lines):

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

When you forget an argument, `help` is faster than a search tab.

The trap

Turning the formatter off because you "like tabs," or sprinkling `# noqa` until `ruff check` is quiet. The team then has two styles and no signal.

The docstring trap is the opposite of silence: a paragraph that repeats the name.

```python
# obvious.py
def add(a, b):
    """Add a and b and return the sum."""
    return a + b


def main():
    print(add(2, 6))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python obvious.py
```

Output:

```
8
```

It works. The docstring adds nothing. Write a docstring when the function hides a rule (`prices cannot be negative`, `status is open|paid`). Skip it when the name and the signature are the documentation.

The boring rule

- Run `uv run ruff format .` before you commit. No debate.
- Run `uv run ruff check .`. Fix `F401` and friends. Do not blanket-ignore.
- Docstrings use `"""triple quotes"""` on the line after `def`.
- One sentence is enough for most functions. `help(fn)` should be readable.
- Do not document `add`. Do document `line_total` if it rejects negative prices.

Try this

1. Break spacing in `ticket_help.py` on purpose. Run `uv run ruff format ticket_help.py` and confirm it snaps back.
2. In `lint_bug.py`, add `import sys` and do not use it. Confirm `ruff check` reports `F401` again, then delete the import.
3. Give `label` a second sentence in the docstring that says the arrow is literal text. Run `ticket_help.py` and read `help()` again.

Dependency Management

Dependency Management

A **dependency** is a package your project imports that did not come with Python. You write it down in **`pyproject.toml`**, lock exact versions in **`uv.lock`**, and install with **`uv sync`**. The boring default is still **the standard library** until a package earns its line.

Mental model

Three files, one job:

File	Role
`pyproject.toml`	What you *want* (`httpx`, `pytest`) and which Python
`uv.lock`	Exact versions that worked, for every machine
`.venv/`	The install of those versions on *this* machine

Commands:

- `uv add pkg` — add a runtime dependency, update the lockfile, install.
- `uv add --dev pkg` — add a tool (`ruff`, `pytest`) that the desk itself does not import.
- `uv lock` — refresh `uv.lock` from `pyproject.toml`.
- `uv sync` — make `.venv` match the lockfile.

Programs in this chapter use **only the standard library** so they run offline. `uv add` is shown as a command you would run when you actually need a package.

Worked examples

Case 1: A complete `pyproject.toml`

Save as `pyproject.toml` in an empty folder. This is enough for a desk that only uses the stdlib:

```toml
# pyproject.toml
[project]
name = "desk"
version = "0.1.0"
requires-python = ">=3.14"
dependencies = []
```

```
[dependency-groups]
dev = [
    "pytest>=9.0",
    "ruff>=0.16",
]
```

`dependencies = []` means runtime imports come from Python itself. The **dev** group is for tools. `uv add --dev pytest` writes that group for you if you would rather not type it.

Pin the interpreter next to it:

```
uv python pin 3.14
uv sync
```

`uv sync` creates `.venv` if needed and installs the dev tools.

Case 2: Stdlib first (runs offline)

Save as `menu.py`. JSON is in the standard library. You do not `uv add json`.

```
# menu.py
import json


def main():
    menu = {"soup": 6, "pie": 9}
    text = json.dumps(menu, indent=2)
    print(text)
    loaded = json.loads(text)
    print(loaded["pie"])


if __name__ == "__main__":
    main()
```

Run:

```
uv run python menu.py
```

Output:

```
{
  "soup": 6,
  "pie": 9
}
9
```

That program does not need the network after Python is installed. Prefer this shape until a library is doing real work you do not want to write.

Case 3: `uv add`, `uv lock`, `uv sync` (documented)

When you *do* need a package — HTTP, a cloud SDK, a parser that is not in the stdlib — you do not edit `dependencies` by guess. You run:

```
uv add httpx
```

`uv add`:

1. Adds `httpx` to `[project] dependencies` in `pyproject.toml`.
2. Resolves a concrete version and writes `uv.lock`.
3. Installs it into `.venv`.

The `dependencies` list then looks like this (the version number will follow whatever `uv add` resolved):

```
dependencies = [
    "httpx>=0.28",
]
```

Related commands:

```
uv lock          # refresh uv.lock from pyproject.toml
uv sync          # install from uv.lock into .venv
uv add --dev ruff pytest
uv remove httpx  # drop a runtime dependency
```

Commit **both** `pyproject.toml` and `uv.lock`. A teammate runs `uv sync` and gets the same versions.

This book's runnable listings stay on the stdlib so you can work through them on a train. When a later chapter truly needs a package, it will say `uv add` and then import it.

Case 4: A lockfile is not a mystery novel

You do not edit `uv.lock` by hand. You read `pyproject.toml`. After `uv add` or a manual edit of `dependencies`, run:

```
uv lock
uv sync
uv run python menu.py
```

`menu.py` still prints the same output as Case 2. Adding a lockfile does not change a program that never imported the new package — which is a reason not to add packages "just in case."

Save as `bill.py` and keep billing in the stdlib:

```
# bill.py
def line_total(qty, price):
    return qty * price


def main():
    print(line_total(2, 6))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python bill.py
```

Output:

```
12
```

The trap

`pip install httpx` into whatever `python3` happens to be, with no `pyproject.toml` and no lockfile. It works on your laptop. It does not work on the next one.

The second trap is adding a package for work the standard library already does: `json`, `pathlib`, `urllib.request`, `datetime`, `subprocess`. Extra dependencies are extra supply chain, extra sync time, and extra version fights. Earn each line in `dependencies`.

The boring rule

- One `pyproject.toml` per project. `requires-python = ">=3.14"`.
- Runtime packages: `uv add pkg`. Tools: `uv add --dev pkg`.
- Commit `uv.lock`. Install with `uv sync`.
- Do not edit `uv.lock` by hand.
- Do not `pip install` into a random interpreter.
- Do not add a package the standard library already covers.

Try this

1. Run `uv add --dev pytest` in a throwaway copy of the folder. Open `pyproject.toml` and find `pytest` under `[dependency-groups]`.

2. In `menu.py`, add a `"tea": 2` entry and confirm `json.dumps` includes it.
3. In `bill.py`, reject a negative `price` with `ValueError` (stdlib only — no new dependency).

Environments and Workspaces

Environments and Workspaces

A **virtual environment** is a folder (usually `.venv`) that holds *this* project's interpreter and packages. `uv run` uses it without a separate "activate" step. The boring default is **one project, one venv**. A **workspace** is several packages sharing one lockfile — skip it until you actually have two installable packages in one repo.

Mental model

Python itself can be installed many times on one machine. A venv points at **one** of those plus the packages `uv sync` installed. Mixing projects in a single global site-packages is how `httpx` versions collide.

`uv` conventions:

- `.venv/` — the environment, local to the project, **not** committed.
- `.python-version` — the pin (`3.14`), **is** committed.
- `uv venv` — create `.venv` explicitly (often unnecessary; `uv sync` and `uv run` create it).
- `uv python pin 3.14` — write `.python-version`.
- `uv run …` — run a command inside `.venv`.

A **workspace** (`[tool.uv.workspace]`) is for a repo that contains more than one package you build. A folder of scripts is not that.

Worked examples

Case 1: Pin, sync, run

In a project folder:

```
uv python pin 3.14
uv sync
```

`.python-version` is:

```
3.14
```

`pyproject.toml` can stay this small:

```
# pyproject.toml
[project]
name = "desk"
version = "0.1.0"
requires-python = ">=3.14"
dependencies = []
```

Save as `where.py`. It checks two things that should be true under `uv run`: the version is 3.14, and the interpreter path lives inside `.venv`.

```
# where.py
import sys
from pathlib import Path


def main():
    parts = Path(sys.executable).parts
    print(sys.version_info[:2])
    print(".venv" in parts)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python where.py
```

Output:

```
(3, 14)
True
```

If the second line is `False`, you are not in the project venv — you ran a bare `python` from somewhere else.

Case 2: `uv venv` is optional

You can create the directory yourself:

```
uv venv
```

That makes `.venv`. Then `uv sync` installs into it, and `uv run python shift.py` uses it. Save as `shift.py`:

```
# shift.py
def main():
    print("shift open")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift.py
```

Output:

```
shift open
```

You *can* "activate" with `source .venv/bin/activate` (or the Windows equivalent) so a bare `python` hits the venv. You do not need to. This book keeps using `uv run` so the command in the chapter matches the command at work.

Case 3: `VIRTUAL_ENV` is set when `uv run` is

Save as `env_flag.py`.

```
# env_flag.py
import os


def main():
    print(bool(os.environ.get("VIRTUAL_ENV")))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python env_flag.py
```

Output:

```
True
```

That flag is how tools notice they are inside an environment. You still do not activate by hand for the examples.

Case 4: Workspaces, briefly

A workspace is a **repo-level** `pyproject.toml` that lists member packages:

```
# pyproject.toml
[project]
name = "desk-root"
version = "0.1.0"
requires-python = ">=3.14"
dependencies = []

[tool.uv.workspace]
members = ["packages/desk", "packages/desk-cli"]
```

Each member has its *own* `pyproject.toml`. One `uv.lock` covers all of them. That layout pays off when `desk-cli` depends on `desk` and you want to change both in one commit.

It does **not** pay off for this book's scripts. Do not create `packages/` because a blog post used the word "monorepo." One folder, one `pyproject.toml`, one `.venv`.

Save as `one_project.py` and keep shipping that idea:

```
# one_project.py
def main():
    print("one project, one venv")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python one_project.py
```

Output:

```
one project, one venv
```

The trap

One "global" environment for every side project, or copying `.venv` between machines (it has absolute paths; it will break). Another: a workspace of empty packages so the repo "looks professional."

Activate scripts, `PYTHONPATH` hacks, and two venvs in one folder (`venv` and `.venv`) are how you run tests against the wrong install. Pick `.venv`, let `uv` own it, put it in `.gitignore`.

The boring rule

- One project directory, one `.venv`, one `.python-version` (`3.14`).
- Create it with `uv sync` or `uv venv`. Run with `uv run`.
- Commit `pyproject.toml`, `uv.lock`, and `.python-version`. Do not commit `.venv`.
- Do not share a venv across projects.
- Do not start a uv workspace until you have two packages that must ship together.
- If `where.py` prints `False` for `.venv`, stop and use `uv run`.

Try this

1. Run `uv run python where.py`, then try `python where.py` without `uv run` if you have another interpreter. Compare the two lines.
2. Delete `.venv` (not `pyproject.toml`). Run `uv sync` then `uv run python shift.py`. Confirm it comes back.
3. Read `.python-version`. Change nothing until a later part asks you to.

Types and Values

Built-in Types

Built-in Types

Every value at the desk is an object of some class. The boring default is to **look** with `type`, **convert on purpose** with `int` / `str` / `float`, and never hope Python will guess.

Mental model

A **value** is an object in memory. A **name** is a label stuck on that object. `type(x)` returns the class of whatever `x` currently points at. `id(x)` is an integer that identifies that object for the life of the process. Use `id` to see whether two names point at the *same* object, not whether they *look* equal.

Built-ins you will use every day:

Type	What it is at the desk
`int`	counts, ticket ids, prices in cents
`float`	measurements (not money)
`bool`	`True` / `False`
`str`	labels, names, status words
`NoneType`	"not set" — one object, spelled `None`

`bool` is a subclass of `int`: `True` behaves like `1` and `False` like `0` if you let it. Do not let it. Treat a bool as yes/no.

Conversion is a function call: `int("12")`, `str(7)`, `float("3.5")`, `bool(0)`. A failed conversion raises `ValueError`. That is better than silent garbage.

Worked examples

Case 1: Look at a ticket

Save as `ticket_types.py`. Print the type of each field so you can see what you actually stored.

```
# ticket_types.py
def main():
    ticket = {
        "id": 7,
        "table": 12,
        "paid": False,
        "note": "window",
```

```
        "closed_at": None,
    }
    for key in ticket:
        value = ticket[key]
        print(key, type(value), value)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_types.py
```

Output:

```
id <class 'int'> 7
table <class 'int'> 12
paid <class 'bool'> False
note <class 'str'> window
closed_at <class 'NoneType'> None
```

`None` is not the string `"None"` and not `False`. It is its own type, with one value.

Case 2: Convert on purpose

The kitchen sends table numbers as text. Prices arrive as strings too. Convert before you add.

```
# convert_order.py
def main():
    table_text = "12"
    cents_text = "1850"
    table = int(table_text)
    cents = int(cents_text)
    print(type(table_text), type(table))
    print(f"table {table}: {cents} cents")
    print(str(cents) + "c")
    print(float(cents) / 100)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python convert_order.py
```

Output:

```
<class 'str'> <class 'int'>
table 12: 1850 cents
1850c
18.5
```

`int("12")` is a new object. The original string is unchanged. `str(cents) + "c"` only works because both sides are strings — adding an `int` to a `str` raises `TypeError`.

Case 3: Equality is not identity

`==` asks "do these look the same?" `id` asks "are these the same object?" Two lists with the same contents are equal and still two objects.

```
# same_object.py
def main():
    a = [7, 12]
    b = a
    c = [7, 12]
    print("a == b", a == b)
    print("a is b", a is b)
    print("a == c", a == c)
    print("a is c", a is c)
    print("same id a,b", id(a) == id(b))
    print("same id a,c", id(a) == id(c))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python same_object.py
```

Output:

```
a == b True
a is b True
a == c True
a is c False
same id a,b True
same id a,c False
```

`b = a` does not copy the list. It sticks a second name on the same list. `id` numbers themselves change every run; comparing them is the useful bit. Use `is` for `None`. Use `==` for values.

Case 4: `bool` is not a number you should add

```
# bool_is_int.py
def main():
    paid = True
    print(type(paid), paid)
    print(paid == 1)
    print(paid + 2)
    print(True + True)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python bool_is_int.py
```

Output:

```
<class 'bool'> True
True
3
2
```

The program is legal. Adding `True` to a count is still a bug. Keep bools in `if` tests and in fields named like `paid`.

The trap

Python will convert in a few places without asking — `if ticket["note"]:` treats `""` as false, and `int(True)` is `1`. It will *not* convert `"12" + 1`. People then reach for implicit tricks (`paid + 1`) or for `id` as a substitute for `==`.

Save as `bad_add.py`:

```
# bad_add.py
def main():
    table = "12"
    print(table + 1)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python bad_add.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "bad_add.py", line 8, in <module>
    main()
  File "bad_add.py", line 4, in main
    print(table + 1)
TypeError: can only concatenate str (not "int") to str
```

The fix is `int(table) + 1` or `table + str(1)`, chosen on purpose. Read the traceback from the bottom.

The boring rule

- Inspect with `type` when a value surprises you.
- Convert with `int`, `str`, `float`, `bool` at the boundary (input, file, kitchen ticket).
- Store money as `int` cents. Use `float` for measurements, not for prices.
- Compare values with `==`. Compare to `None` with `is`.
- Do not add bools. Do not use `id` as a hash you print in logs.

Try this

1. In `convert_order.py`, feed `cents_text = "18.50"` and catch `ValueError`. Print a clear message. Then parse it with `float` and convert to cents with `round`.
2. In `ticket_types.py`, add a `covers` field as the string `"4"`. Convert it to `int` before you print.
3. In `same_object.py`, append `99` to `b` and print `a` and `c`. Notice which list changed.

None, Truthiness, and Initialization

None, Truthiness, and Initialization

`None` means "not set." `False`, `0`, and `[]` are set — they just happen to be empty or no. The boring default is to **test for the value you mean**, not for "looks false in an `if`."

Mental model

Python has a small set of values that are **falsy** in a boolean context: `None`, `False`, `0`, `0.0`, `""`, `[]`, `{}`, `set()`, and a few others. Everything else is **truthy**, including `"0"`, `"False"`, and `[0]`.

That list is a convenience for `if items:` meaning "is there anything here?" It is a trap when `0` is a legal table number, when `[]` is a legal empty note list, or when `None` means "the cook has not answered yet."

Initialization is how a function gets a starting container. A default argument is evaluated **once**, when the `def` line runs, not each call. A mutable default (`[]`, `{}`) is shared across calls. That is the classic Python footgun.

Worked examples

Case 1: Four different "no"s

Save as `four_nos.py`. A missing close time, an unpaid check, a zero-cover walk-in, and an empty note list are not the same fact.

```
# four_nos.py
def describe(ticket):
    if ticket["closed_at"] is None:
        print("still open")
    if ticket["paid"] is False:
        print("not paid")
    if ticket["covers"] == 0:
        print("walk-in, covers unknown")
    if ticket["notes"] == []:
        print("no notes yet")


def main():
    ticket = {
        "closed_at": None,
```

```
        "paid": False,
        "covers": 0,
        "notes": [],
    }
    describe(ticket)
    print("bool(None)", bool(None))
    print("bool(False)", bool(False))
    print("bool(0)", bool(0))
    print("bool([])", bool([]))
    print("None == False", None == False)
    print("None == 0", None == 0)
    print("False == 0", False == 0)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python four_nos.py
```

Output:

```
still open
not paid
walk-in, covers unknown
no notes yet
bool(None) False
bool(False) False
bool(0) False
bool([]) False
None == False False
None == 0 False
False == 0 True
```

All four are falsy. Only `False == 0` is actually equal, because `bool` subclasses `int`. `None` equals neither. Test `is None`, `is False`, `== 0`, and `== []` when those meanings matter.

Case 2: `if notes:` is fine for "any notes"

When empty really means "skip," a truthiness check is the boring line.

```
# any_notes.py
def print_notes(notes):
    if notes:
        print("notes:", ", ".join(notes))
```

```
    else:
        print("no notes")


def main():
    print_notes(["window", "allergy"])
    print_notes([])


if __name__ == "__main__":
    main()
```

Run:

```
uv run python any_notes.py
```

Output:

```
notes: window, allergy
no notes
```

That is the intended use. Do not use `if notes:` when `notes` might be `None` (not loaded) versus `[]` (loaded, empty). Those two states need different handling.

Case 3: Default `None`, then make a list

When a function should start with an empty list unless the caller passes one, default to `None` and create the list inside.

```
# add_note.py
def add_note(note, notes=None):
    if notes is None:
        notes = []
    notes.append(note)
    return notes


def main():
    a = add_note("window")
    b = add_note("allergy")
    shared = []
    add_note("booth", shared)
    add_note("quiet", shared)
    print(a)
    print(b)
    print(shared)
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python add_note.py
```

Output:

```
['window']
['allergy']
['booth', 'quiet']
```

Each call that omits `notes` gets a **new** list. The caller who passes `shared` keeps one list on purpose.

The trap

A mutable default argument is evaluated once. Every call that omits the argument appends to the **same** list.

```
# sticky_notes.py
def add_note(note, notes=[]):
    notes.append(note)
    return notes


def main():
    first = add_note("window")
    second = add_note("allergy")
    print("first ", first)
    print("second", second)
    print("same object", first is second)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python sticky_notes.py
```

Output:

```
first  ['window', 'allergy']
second ['window', 'allergy']
same object True
```

`first` and `second` are one list. The second ticket inherited the first ticket's note. The fix is Case 3: default `None`, assign `notes = []` inside the function.

The same trap hits `def label(ticket, extra={}):` — one dict, shared forever. Default to `None` for every mutable.

The boring rule

- `None` means missing. Test with `is None` / `is not None`.
- `0`, `[]`, `""`, and `False` are real values. Test them by name when `0` or empty is legal.
- `if items:` is fine when you only care "any vs none."
- Never default an argument to `[]` or `{}`. Use `None` and create inside.
- Do not write `if not ticket["closed_at"]` if `closed_at` could be `None` or `0` for different reasons.

Try this

1. Change `four_nos.py` so `covers` is `None` (unknown) versus `0` (walk-in). Print two different sentences.
2. In `add_note.py`, add a third call `add_note("vip")` and confirm you still get a fresh list.
3. Copy `sticky_notes.py` and default `notes=None` instead. Re-run. `first is second` should be `False`.

Literals, Numbers, and Bytes

Literals, Numbers, and Bytes

A **literal** is a value written in the source: `12`, `1_850`, `0xFF`, `"window"`, `b"OK"`. The boring default is to write numbers so a human can read them, keep money in cents or `Decimal`, and treat **bytes** as raw and **str** as text.

Mental model

Python reads an integer in decimal unless you mark a base: `0x` hex, `0o` octal, `0b` binary. Underscores in numbers are ignored: `1_850` is `1850`. They exist so a price or a mask is skimmable.

`int` division with `//` stays an `int`. `/` always gives a `float`. `float` is binary floating point. `0.1 + 0.2` is not `0.3`. For money, store **integer cents** or use `decimal.Decimal` with a string constructor.

A **str** is Unicode text. **bytes** is a sequence of 0–255. You move between them with `encode` and `decode` and an encoding name, almost always `"utf-8"`. Adding a `str` to `bytes` raises `TypeError`.

Text formatting: an **f-string** (`f"table {n}"`) is the boring default. `str.format` still works. `%` formatting is legacy.

Worked examples

Case 1: Readable numbers and bases

Save as `readable_numbers.py`.

```
# readable_numbers.py
def main():
    cents = 1_850
    seats = 10_000
    flags = 0xA5
    print(cents, seats, flags)
    print(hex(flags), bin(flags))
    print(12 / 5)
    print(12 // 5)
    print(12 % 5)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python readable_numbers.py
```

Output:

```
1850 10000 165
0xa5 0b10100101
2.4
2
2
```

`0xA5` is 165. `/` is float division even when both sides are ints. `//` is floor division. `%` is remainder.

Case 2: Float is the wrong type for a check

```
# float_check.py
from decimal import Decimal


def main():
    print(0.1 + 0.2)
    print(Decimal("0.1") + Decimal("0.2"))
    soup = 850
    tea = 400
    print("cents", soup + tea)
    print("dollars", (soup + tea) / 100)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python float_check.py
```

Output:

```
0.30000000000000004
0.3
cents 1250
dollars 12.5
```

Add cents as ints. Convert to dollars only when you print. If you need exact decimal fractions, construct `Decimal` from a **string**, not from a float (`Decimal(0.1)` already has the error).

Case 3: `str` is text, `bytes` is raw

```
# label_bytes.py
def main():
    label = "café table 12"
    raw = label.encode("utf-8")
    print(type(label), label)
    print(type(raw), raw)
    print(list(raw))
    back = raw.decode("utf-8")
    print(back)
    print(label == back)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python label_bytes.py
```

Output:

```
<class 'str'> café table 12
<class 'bytes'> b'caf\xc3\xa9 table 12'
[99, 97, 102, 195, 169, 32, 116, 97, 98, 108, 101, 32, 49, 50]
café table 12
True
```

`é` is two bytes in UTF-8 (`195`, `169`). The `b'...'` display is Python's way of showing those bytes. Decode with the same encoding you encoded with.

Case 4: f-strings versus `format`

```
# format_ticket.py
def main():
    table = 12
    cents = 1850
    print(f"table {table}: {cents} cents")
    print("table {}: {} cents".format(table, cents))
    print("table {t}: {c} cents".format(t=table, c=cents))
    print(f"table {table:>4}: ${cents / 100:.2f}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python format_ticket.py
```

Output:

```
table 12: 1850 cents
table 12: 1850 cents
table 12: 1850 cents
table   12: $18.50
```

Prefer the f-string. Use format specs (`:>4`, `:.2f`) inside the braces. Keep `str.format` when the template itself is a string you loaded from a file.

The trap

Mixing `str` and `bytes`, or using `float` as a running total for money, both fail in ways that look like "Python is weird" until you look at the types.

```
# mix_text.py
def main():
    header = b"TICKET"
    name = "12"
    print(header + name)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python mix_text.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "mix_text.py", line 9, in <module>
    main()
  File "mix_text.py", line 5, in main
    print(header + name)
TypeError: can't concat str to bytes
```

The fix is one side: `header + name.encode("utf-8")` or `header.decode("utf-8") + name`. Pick text or raw and convert at the edge (socket, file, subprocess).

The boring rule

- Write `1_850`, not `1850`, when the extra marks help a human.
- Use `0x` / `0b` when the value *is* a bit pattern. Do not decorate ordinary counts.
- Money: integer cents, or `Decimal("18.50")`. Never accumulate `float` dollars.
- `str` in the program. `bytes` at the wire. Encode and decode with `"utf-8"` unless a protocol names another encoding.
- f-strings for almost all formatting.

Try this

1. In `readable_numbers.py`, print `0b1010` and `0o12` and confirm both are ten.
2. Change `float_check.py` so a 10% service charge is applied in cents with integer arithmetic (`cents * 10 // 100`).
3. In `label_bytes.py`, decode with `"latin-1"` instead of `"utf-8"` and print the result. Put `"utf-8"` back.

Names, Scope, and LEGB

Names, Scope, and LEGB

A name in Python is looked up in a fixed order: **Local, Enclosing, Global, Built-in** (LEGB). The boring default is to **pass values in and return values out**. `global` and `nonlocal` exist; reach for them when a nested function must rebind a name, not as a way to share a desk-wide bag of state.

Mental model

- **Local:** names assigned in the current function.
- **Enclosing:** names in an outer function, if this `def` is nested.
- **Global:** names assigned at module level (the file).
- **Built-in:** names Python already has (`len`, `list`, `str`, `print`).

Assignment makes a name local to that function, **for the whole function**, even if the `=` sits below a read. That is why `count = count + 1` inside a function does not quietly update a module-level `count` unless you declared `global count`.

`global name` means "this assignment writes the module-level name." `nonlocal name` means "this assignment writes the nearest enclosing function's name." Both are rebinding. Mutating a list you received as an argument does not need either keyword — you are not assigning to the name, you are changing the object.

Worked examples

Case 1: Local hides global

Save as `shift_count.py`. The function's `open_tickets` is a new name. The module-level one is untouched.

```
# shift_count.py
open_tickets = 3


def start_shift():
    open_tickets = 0
    print("inside", open_tickets)


def main():
```

```
    start_shift()
    print("module", open_tickets)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_count.py
```

Output:

```
inside 0
module 3
```

That is ordinary. The inner assignment never writes the outer name.

Case 2: `global` when you truly rebind module state

A flag at module level is sometimes real (a script-sized program). Declare it.

```
# desk_open.py
desk_open = False


def open_desk():
    global desk_open
    desk_open = True


def main():
    print("before", desk_open)
    open_desk()
    print("after", desk_open)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_open.py
```

Output:

```
before False
after True
```

Without `global`, `open_desk` would create a local `desk_open` and the module flag would stay `False`. Prefer returning a new value in anything larger than a script: `desk_open = open_desk(desk_open)`.

Case 3: `nonlocal` for a nested counter

```python
# ticket_ids.py
def make_id_factory(start):
    next_id = start

    def next_ticket_id():
        nonlocal next_id
        next_id += 1
        return next_id

    return next_ticket_id


def main():
    next_id = make_id_factory(100)
    print(next_id())
    print(next_id())
    print(next_id())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_ids.py
```

Output:

```
101
102
103
```

`next_id += 1` is assignment, so Python would treat `next_id` as local to `next_ticket_id` unless `nonlocal` says otherwise. The enclosing `make_id_factory` keeps the counter. Two factories would keep two counters.

Case 4: Mutate without `global`

```
# queue_append.py
queue = []


def land(ticket):
    queue.append(ticket)


def main():
    land({"id": 7})
    land({"id": 8})
    print(queue)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python queue_append.py
```

Output:

```
[{'id': 7}, {'id': 8}]
```

`queue.append` does not assign to the name `queue`. LEGB finds the global list and mutates it. This works and it is still a slippery style: any function can change the queue. Passing the list in is clearer.

The trap

Shadowing a built-in. The name `list` in the local (or global) scope wins over the built-in constructor.

```
# shadow_list.py
def tables_on_shift(raw):
    list = raw.split(",")
    return list(int(t) for t in list)


def main():
    print(tables_on_shift("3,11,12"))
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python shadow_list.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "shadow_list.py", line 12, in <module>
    main()
  File "shadow_list.py", line 8, in main
    print(tables_on_shift("3,11,12"))
  File "shadow_list.py", line 4, in tables_on_shift
    return list(int(t) for t in list)
TypeError: 'list' object is not callable
```

`list = raw.split(",")` rebound the name. The next line tried to call a list. The fix is a real name: `parts = raw.split(",")`, then `return [int(t) for t in parts]`. Do not name variables `list`, `str`, `id`, `type`, `len`, or `dict`.

A related trap: reading a global, then assigning to the same name later in the function. Python marks it local for the whole body and the early read raises `UnboundLocalError`.

The boring rule

- Pass arguments. Return results. That is the default.
- Use `global` only for true module-level rebinding in a small script.
- Use `nonlocal` when a nested function must rebind an enclosing name (counters, small factories).
- Mutating a list or dict through a name does not need `global`. Still prefer passing the object in.
- Never shadow built-ins.

Try this

1. Remove `global desk_open` from `desk_open.py` and run. Then print `desk_open` inside `open_desk` *before* the assignment and read the `UnboundLocalError`.
2. In `ticket_ids.py`, make two factories (`make_id_factory(100)` and `make_id_factory(500)`) and print a few ids from each.
3. Fix `shadow_list.py` with a name that is not `list`. Run it.

Control Flow

Conditionals and match

Conditionals and match

Control flow at the desk is mostly "if this status, do that." The boring default is a short `if` / `elif` / `else`, a **guard clause** that returns early, and `match` / `case` when you are dispatching on a closed set of ticket statuses. A ternary (`x if cond else y`) is an expression. Use it for a tiny choice, not for a paragraph.

Mental model

`if` tests a condition and runs a block. `elif` is another test, only if the previous tests failed. `else` runs when none of them matched. Python has no `switch` statement. Since 3.10 it has `match`, which compares a **value** against **patterns**.

A **guard clause** is an `if` at the top of a function that returns (or raises) so the rest of the function can assume the happy path. That beats nesting `if paid: if table: if notes:` five layers deep.

`match ticket["status"]:` with `case "open":` is pattern matching on a string. `case _:` is the default. You can also match mapping patterns (`case {"status": "open", "table": n}:`) when the shape is the point. Keep patterns flat. A `match` that needs a comment to explain the pattern should have been an `if`.

The ternary `label = "paid" if ticket["paid"] else "due"` is fine. `print("paid" if ticket["paid"] else "due" if ticket["open"] else "void")` is not.

Worked examples

Case 1: `if` / `elif` / `else` on a check

Save as `check_status.py`.

```
# check_status.py
def describe(ticket):
    if ticket["paid"]:
        return "closed"
    elif ticket["cents"] == 0:
        return "comped"
    else:
        return "due"
```

```
def main():
    tickets = [
        {"paid": True, "cents": 1850},
        {"paid": False, "cents": 0},
        {"paid": False, "cents": 400},
    ]
    for t in tickets:
        print(describe(t))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python check_status.py
```

Output:

```
closed
comped
due
```

Order matters. A paid zero-cent ticket is `"closed"` because that test runs first. Put the most specific real-world rule first, then the leftovers.

Case 2: Guard clauses, then the work

```
# seat.py
def seat(ticket):
    if ticket is None:
        raise ValueError("no ticket")
    if ticket["table"] <= 0:
        raise ValueError(f"table {ticket['table']}: must be positive")
    if ticket["status"] != "open":
        return f"ticket {ticket['id']} not seated"
    return f"seated ticket {ticket['id']} at table {ticket['table']}"


def main():
    print(seat({"id": 7, "table": 12, "status": "open"}))
    print(seat({"id": 8, "table": 4, "status": "paid"}))
    try:
        seat({"id": 9, "table": 0, "status": "open"})
    except ValueError as e:
```

```
        print(e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python seat.py
```

Output:

```
seated ticket 7 at table 12
ticket 8 not seated
table 0: must be positive
```

The happy path is one `return` at the bottom. Bad input raises. A legal-but-not-seatable ticket returns a string. Do not nest those three decisions.

Case 3: `match` on ticket status

```
# route_ticket.py
def route(ticket):
    match ticket["status"]:
        case "open":
            return f"send ticket {ticket['id']} to kitchen"
        case "fired":
            return f"ticket {ticket['id']} is on the pass"
        case "paid":
            return f"ticket {ticket['id']} is closed"
        case "void":
            return f"ticket {ticket['id']} was voided"
        case _:
            raise ValueError(f"unknown status {ticket['status']!r}")


def main():
    for status in ("open", "fired", "paid", "void"):
        print(route({"id": 7, "status": status}))
    try:
        route({"id": 7, "status": "lost"})
    except ValueError as e:
        print(e)


if __name__ == "__main__":
```

```
    main()
```

Run:

```
uv run python route_ticket.py
```

Output:

```
send ticket 7 to kitchen
ticket 7 is on the pass
ticket 7 is closed
ticket 7 was voided
unknown status 'lost'
```

`case _:` is required if you want a default. Leaving it off makes a non-match a silent no-op, which is how statuses disappear. Raise on unknown.

Case 4: A ternary, kept small

```
# due_label.py
def due_label(cents):
    return "even" if cents == 0 else f"{cents} due"


def main():
    print(due_label(0))
    print(due_label(400))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python due_label.py
```

Output:

```
even
400 due
```

That is the whole point of a ternary: an expression that picks one of two values. The moment you want a third branch, write `if` / `elif` / `else`.

The trap

A chain of `if`s that are not `elif` will run more than one branch. A `match` without `case _:` will drop unknown statuses on the floor.

```
# overlapping.py
def tags(ticket):
    labels = []
    if ticket["cents"] > 0:
        labels.append("has-total")
    if ticket["paid"]:
        labels.append("paid")
    else:
        labels.append("unpaid")
    return labels


def main():
    print(tags({"cents": 400, "paid": True}))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python overlapping.py
```

Output:

```
['has-total', 'paid']
```

Here two `if`s are correct: paid-ness is independent of the total. The trap is using that shape when you meant **one** outcome. If a ticket should be exactly one of `due` / `comped` / `closed`, use `elif` or `match`, not stacked `if`s.

The boring rule

- One outcome → `if` / `elif` / `else` or `match`.
- Independent flags → separate `if`s.
- Guard clauses at the top; happy path last.
- `match` on a closed set of statuses. Always handle `_` (raise or log).
- Ternary only for a two-value expression.

Try this

1. Add a `"held"` status to `route_ticket.py` that returns `"ticket {id} is held at the bar"`.
2. In `seat.py`, guard on missing `"table"` with `if "table" not in ticket` and raise `ValueError`.
3. Rewrite `due_label` with `if` / `else` instead of a ternary. Keep the same output.

Loops and Iteration

Loops and Iteration

A loop walks a collection or repeats until a condition fails. The boring default is `for item in items`. Use `enumerate` when you need a position, `zip` when you need two lists in lockstep, `range` when you need integers, and `while` when the stop condition is not "the list ended."

Mental model

`for x in iterable:` binds `x` to each element. A **list**, **tuple**, **str**, **dict** (keys), and **range** are all iterable. `range(n)` produces `0 .. n-1`. `range(start, stop)` stops *before* `stop`.

`enumerate(items, start=0)` yields `(index, item)`. `zip(a, b)` yields pairs and stops at the shorter input.

`break` leaves the loop. `continue` skips to the next iteration. A loop may have an `else:` clause that runs **if the loop did not `break`**. That is real Python. It is also easy to misread as "run after the loop." The boring default is: **do not use `for/else`**. Use a flag or return.

`while` is for "keep going until the pass is empty," not for walking a list you already have.

Worked examples

Case 1: `for`, `range`, `enumerate`

Save as `walk_tickets.py`.

```
# walk_tickets.py
def main():
    tickets = ["open", "fired", "paid"]
    for status in tickets:
        print("status", status)
    for i in range(len(tickets)):
        print("index", i, tickets[i])
    for i, status in enumerate(tickets, start=1):
        print(f"#{i} {status}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python walk_tickets.py
```

Output:

```
status open
status fired
status paid
index 0 open
index 1 fired
index 2 paid
#1 open
#2 fired
#3 paid
```

Prefer `for status in tickets` when you do not need the index. Prefer `enumerate` over `range(len(...))` when you do. `range(len)` is legal and usually worse.

Case 2: `zip` two columns

```
# zip_tables.py
def main():
    tables = [3, 11, 12]
    covers = [2, 4, 2]
    for table, n in zip(tables, covers):
        print(f"table {table}: {n} covers")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python zip_tables.py
```

Output:

```
table 3: 2 covers
table 11: 4 covers
table 12: 2 covers
```

If one list is shorter, `zip` stops early and does not warn. Check lengths when the lists must match: `if len(tables) != len(covers): raise ValueError(...)`.

Case 3: `while`, `break`, `continue`

```
# pass_window.py
def main():
    pass_window = ["soup", "SKIP", "tea", "STOP", "cake"]
    while pass_window:
        item = pass_window.pop(0)
        if item == "SKIP":
            continue
        if item == "STOP":
            break
        print("plate", item)
    print("left", pass_window)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python pass_window.py
```

Output:

```
plate soup
plate tea
left ['cake']
```

`continue` drops `SKIP`. `break` leaves `cake` on the window. `pop(0)` is simple here; a real queue would use `collections.deque`. The `while pass_window:` test is "still something there."

Case 4: Loop `else` — then do not make it the default

Python runs the `else` on a loop when no `break` happened. This program finds a free table.

```
# find_table.py
def find_free(tables, want):
    for table in tables:
        if table["n"] == want and table["free"]:
            print(f"seated at {want}")
            break
    else:
        print(f"no free table {want}")


def main():
```

```
    tables = [
        {"n": 11, "free": False},
        {"n": 12, "free": True},
    ]
    find_free(tables, 12)
    find_free(tables, 3)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python find_table.py
```

Output:

```
seated at 12
no free table 3
```

It works. Readers still trip over `else` hanging off `for`. The boring version is a function that **returns** the table or `None`, then the caller prints. Use loop `else` only when the team already reads it without blinking. This book's default is to avoid it.

```
# find_table_return.py
def find_free(tables, want):
    for table in tables:
        if table["n"] == want and table["free"]:
            return table
    return None


def main():
    tables = [
        {"n": 11, "free": False},
        {"n": 12, "free": True},
    ]
    for want in (12, 3):
        found = find_free(tables, want)
        if found is None:
            print(f"no free table {want}")
        else:
            print(f"seated at {want}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python find_table_return.py
```

Output:

```
seated at 12
no free table 3
```

Same results. The `else` belongs to `if`, where everyone looks for it.

The trap

Mutating a list while you `for` over it. Inserts and deletes shift the remaining items; you skip rows or Python raises.

```
# mutate_while.py
def main():
    tickets = ["open", "void", "open", "paid"]
    for status in tickets:
        if status == "void":
            tickets.remove(status)
    print(tickets)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python mutate_while.py
```

Output:

```
['open', 'open', 'paid']
```

This tiny list happens to survive. A second `"void"` next to the first often does not get visited. Build a new list instead:

```
# keep_tickets.py
def main():
    tickets = ["open", "void", "open", "paid"]
    kept = [s for s in tickets if s != "void"]
    print(kept)
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python keep_tickets.py
```

Output:

```
['open', 'open', 'paid']
```

The boring rule

- `for item in items` is the default loop.
- `enumerate` for positions. `zip` for parallel lists. `range` for integers.
- `while` when the end is a condition, not a length.
- `break` / `continue` are fine when they make a guard, not a maze.
- Do not use `for`/`else` as the house style. Return or set a flag.
- Do not mutate the list you are iterating. Make a new one.

Try this

1. In `zip_tables.py`, add a fourth table without a cover count. Print `len` of both lists and raise `ValueError` if they differ.
2. Rewrite `pass_window.py` with a `for` loop over a copy of the list (`for item in list(pass_window):`) and `break`/`continue` the same way. Compare what is left.
3. Change `find_table_return.py` so it prints the whole found dict, not only the number.

Comprehensions and Walrus

Comprehensions and Walrus

A **comprehension** builds a list, dict, or set from an iterable in one expression. A **generator expression** does the same walk without building the collection up front. The **walrus operator** `:=` assigns inside an expression. The boring default is a **one-line filter or map** you can read aloud. Nested comprehensions and walrus-in-every-`if` are how a desk script becomes a crossword.

Mental model

```
[expression for item in items if condition]
```

That is a loop and an optional filter, producing a list. Change `[]` to `{}` with `key: value` for a dict, or `{expression}` for a set. Parentheses `(expression for item in items)` make a **generator expression**: lazy, one item at a time, useful in `sum`, `any`, `all`.

`:=` names a value in the middle of an `if` or `while` so you do not compute it twice. `if (n := len(tickets)) > 10:` binds `n` and tests it. The parentheses are required in an `if`.

Comprehensions are not a second language. If the expression needs two `for`s, a function call with side effects, or a comment, write a loop.

Worked examples

Case 1: List comprehension as a filter

Save as `open_ids.py`.

```
# open_ids.py
def main():
    tickets = [
        {"id": 7, "status": "open", "cents": 1850},
        {"id": 8, "status": "paid", "cents": 400},
        {"id": 9, "status": "open", "cents": 0},
    ]
    open_ids = [t["id"] for t in tickets if t["status"] == "open"]
    print(open_ids)


if __name__ == "__main__":
```

```
    main()
```

Run:

```
uv run python open_ids.py
```

Output:

```
[7, 9]
```

Read it: "the id for each ticket if the status is open." That sentence is the test. If you cannot say it, do not write it as a comprehension.

Case 2: Dict and set comprehensions

```
# index_tables.py
def main():
    tickets = [
        {"id": 7, "table": 12},
        {"id": 8, "table": 3},
        {"id": 9, "table": 12},
    ]
    by_id = {t["id"]: t["table"] for t in tickets}
    tables = {t["table"] for t in tickets}
    print(by_id)
    print(sorted(tables))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python index_tables.py
```

Output:

```
{7: 12, 8: 3, 9: 12}
[3, 12]
```

Dict keys must be unique: a second `7` would overwrite. The set drops the duplicate table `12`. Set print order is not a promise, so this program sorts before printing.

Case 3: Generator expression for a total

```
# sum_open.py
def main():
    tickets = [
        {"status": "open", "cents": 1850},
        {"status": "paid", "cents": 400},
        {"status": "open", "cents": 250},
    ]
    total = sum(t["cents"] for t in tickets if t["status"] == "open")
    print(total)
    any_void = any(t["status"] == "void" for t in tickets)
    print(any_void)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python sum_open.py
```

Output:

```
2100
False
```

`sum(...)` takes a generator expression without extra parentheses when it is the only argument. You do not keep a list of cents you will never use again.

Case 4: Walrus to name a length once

```
# cover_cap.py
def main():
    covers = [2, 4, 2, 5, 3]
    if (n := len(covers)) > 4:
        print(f"{n} parties waiting")
    else:
        print(f"{n} parties, under cap")
    while (n := len(covers)) > 3:
        covers.pop()
        print("sat one,", n - 1, "left")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python cover_cap.py
```

Output:

```
5 parties waiting
sat one, 4 left
sat one, 3 left
```

The `if` uses `n` in the message. The `while` re-reads the length each pass. That is the honest use: **compute, name, test**, without a duplicate `len(covers)`.

The trap

Nested comprehensions that zip the whole desk into one line. They run. Nobody wants to edit them on a Saturday.

```
# too_clever_comp.py
def main():
    shifts = [
        {"day": "mon", "tickets": [{"id": 1, "ok": True}, {"id": 2, "ok": False}]},
        {"day": "tue", "tickets": [{"id": 3, "ok": True}]},
    ]
    ids = [
        t["id"]
        for shift in shifts
        if shift["day"] != "sun"
        for t in shift["tickets"]
        if t["ok"]
    ]
    print(ids)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python too_clever_comp.py
```

Output:

```
[1, 3]
```

The two `for`s are a nested loop with the outer `for` first. That is the opposite of how people read English nested phrases. The boring fix is a loop, or a small function:

```
# ok_ids.py
def ok_ids(shifts):
    ids = []
    for shift in shifts:
        if shift["day"] == "sun":
            continue
        for t in shift["tickets"]:
            if t["ok"]:
                ids.append(t["id"])
    return ids


def main():
    shifts = [
        {"day": "mon", "tickets": [{"id": 1, "ok": True}, {"id": 2, "ok": False}]},
        {"day": "tue", "tickets": [{"id": 3, "ok": True}]},
    ]
    print(ok_ids(shifts))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ok_ids.py
```

Output:

```
[1, 3]
```

Same ids. You can put a breakpoint on `continue`. Nested comps also hide side effects (`print`, appends to an outer list). Do not put those in the expression.

Walrus has a twin trap: `if n := len(covers) > 4` without parentheses binds `n` to `True`/`False`, not the length. Always parenthesize `:=` in `if`.

The boring rule

- One `for`, one optional `if`, a cheap expression: comprehension.
- Totals and any/all: generator expression.
- Two or more `for`s: nested loops or a helper function.
- `:=` when it removes a duplicate call, with parentheses in `if`.
- No side effects inside a comprehension.

Try this

1. In `open_ids.py`, also build a set of open statuses with a set comprehension. It should print `{'open'}`.
2. Add a `"void"` ticket to `sum_open.py` and confirm `any_void` becomes `True`.
3. Rewrite `cover_cap.py`'s `if` without `:=` using `n = len(covers)` on the previous line. Keep the output.

Collections

Lists and Tuples

Lists and Tuples

A **list** is a growable sequence. A **tuple** is a fixed sequence. The boring default is a **list** when the desk will append, sort, or drop rows, and a **tuple** when the values are a **record** (id, table, status) that should not grow a fourth surprise field by accident.

Mental model

Both are ordered, both index from `0`, both slice with `items[start:stop]`. The difference is mutability. `tickets.append(t)` changes the list. There is no `tuple.append`. To "change" a tuple you build a new one.

Names are not copies. `b = a` makes two names for one list. `a[:]` and `list(a)` make a **shallow copy**: a new outer list, same inner objects. Nested lists still alias. `copy.deepcopy` copies the tree.

A slice of a list is a new list. A slice of a tuple is a new tuple. Either way, the *elements* are the same objects.

A tuple as a record is positional: `ticket[0]` is easy to mix up with `ticket[1]`. Unpack it (`id, table, status = ticket`) or move to a dataclass when the record grows names.

Worked examples

Case 1: Lists change, tuples do not

Save as `list_vs_tuple.py`.

```
# list_vs_tuple.py
def main():
    tickets = [7, 8]
    tickets.append(9)
    tickets[0] = 70
    print(tickets)
    row = (7, 12, "open")
    print(row)
    print(row[0], row[-1])
    try:
        row[0] = 70
    except TypeError as e:
        print(e)
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python list_vs_tuple.py
```

Output:

```
[70, 8, 9]
(7, 12, 'open')
7 open
'tuple' object does not support item assignment
```

`row[-1]` is the last field. Use it. Do not mutate a tuple; you cannot.

Case 2: Alias versus copy

```
# alias_copy.py
def main():
    a = [7, 8, 9]
    alias = a
    copied = a[:]
    alias.append(10)
    copied.append(99)
    print("a     ", a)
    print("alias ", alias)
    print("copied", copied)
    print("alias is a", alias is a)
    print("copied is a", copied is a)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python alias_copy.py
```

Output:

```
a      [7, 8, 9, 10]
alias  [7, 8, 9, 10]
copied [7, 8, 9, 99]
```

```
alias is a True
copied is a False
```

`alias.append` changed `a`. The slice did not. When a function does `tickets.append(...)`, every name that points at that list sees the append.

Case 3: Shallow copy and nested lists

```python
# shallow.py
import copy


def main():
    shift = [[7, 8], [9]]
    shallow = shift[:]
    deep = copy.deepcopy(shift)
    shallow[0].append(70)
    print("shift  ", shift)
    print("shallow", shallow)
    print("deep   ", deep)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shallow.py
```

Output:

```
shift   [[7, 8, 70], [9]]
shallow [[7, 8, 70], [9]]
deep    [[7, 8], [9]]
```

`shift[:]` copied the outer list only. `shallow[0]` is the same inner list as `shift[0]`. `deepcopy` made new inner lists. Use deepcopy when you have lists of lists (or lists of dicts) and you need a snapshot.

Case 4: Tuple as a record

```python
# ticket_row.py
def label(row):
    ticket_id, table, status = row
    return f"ticket {ticket_id} table {table} ({status})"
```

```
def main():
    open_row = (7, 12, "open")
    paid_row = (8, 3, "paid")
    print(label(open_row))
    print(label(paid_row))
    print(open_row + (False,))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_row.py
```

Output:

```
ticket 7 table 12 (open)
ticket 8 table 3 (paid)
(7, 12, 'open', False)
```

Unpacking names the fields for the length of the function. `+` on tuples concatenates and returns a **new** tuple; `open_row` is unchanged. A fourth anonymous `False` is why records outgrow tuples — you cannot tell what `False` is. That is the cue for a dataclass.

The trap

Slicing looks like a copy of everything. It is not, once values are mutable.

```
# slice_alias.py
def main():
    tickets = [{"id": 7, "status": "open"}, {"id": 8, "status": "paid"}]
    window = tickets[:1]
    window[0]["status"] = "void"
    print("window ", window)
    print("tickets", tickets)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python slice_alias.py
```

Output:

```
window  [{'id': 7, 'status': 'void'}]
tickets [{'id': 7, 'status': 'void'}, {'id': 8, 'status': 'paid'}]
```

`window` is a new list of length 1. The dict inside is the same dict as `tickets[0]`. Changing `status` through either name changes both. Copy the dict (`dict(t)` or `{**t}`) or `deepcopy` the structure when the window must be independent.

The boring rule

- List: variable length, same kind of thing (tickets, tables, notes).
- Tuple: fixed record or a return of two or three values. Unpack it.
- `b = a` aliases. `a[:]` / `list(a)` shallow-copy. Nested mutables still shared.
- Do not assign through a slice you thought was a snapshot.
- When a tuple grows named fields, use a dataclass.

Try this

1. In `alias_copy.py`, use `list(a)` instead of `a[:]`. Confirm `copied is a` is still `False`.
2. In `ticket_row.py`, unpack with a wrong number of names (`id, table = open_row`) and read the `ValueError`.
3. Fix `slice_alias.py` so the window holds `{**tickets[0]}` (a new dict). Void the window copy only.

Strings and Bytes

Strings and Bytes

A **str** is immutable Unicode text. **bytes** is immutable raw 8-bit data. The boring default is **text inside the program**, **bytes at the edge** (files, sockets, subprocesses), and `encode` / `decode` with `"utf-8"` when you cross that edge.

Mental model

You cannot change a character in a string in place. `label.replace("12", "3")` returns a **new** string. The name `label` still points at the old one unless you assign the result back.

`split` turns a string into a list of strings. `join` goes the other way: `", ".join(parts)`. The joiner is the string you call `join` on. `split` with no argument splits on any whitespace and drops empties. `split(",")` keeps empties (`"3,,12"` → `['3', '', '12']`).

f-strings are expressions in braces. They call `str()` on the value unless you add a conversion (`!r` for `repr`). Format specs go after a colon: `{table:>4}`, `{cents / 100:.2f}`.

`encode` is `str → bytes`. `decode` is `bytes → str`. Both take an encoding and an error mode (`"strict"` default, `"replace"`, `"ignore"`). Mismatched encodings produce mojibake or `UnicodeDecodeError`.

Worked examples

Case 1: Immutable text, assign the result

Save as `rewrite_label.py`.

```
# rewrite_label.py
def main():
    label = "table 12"
    label.replace("12", "3")
    print("after replace, no assign:", label)
    label = label.replace("12", "3")
    print("after assign:", label)
    print(label.upper())
    print("table" in label)


if __name__ == "__main__":
```

```
    main()
```

Run:

```
uv run python rewrite_label.py
```

Output:

```
after replace, no assign: table 12
after assign: table 3
TABLE 3
True
```

Forgetting to assign `replace` / `upper` / `strip` is the daily string bug. Methods return new objects.

Case 2: `split` and `join`

```
# split_tables.py
def parse_tables(raw):
    parts = raw.split(",")
    return [int(p.strip()) for p in parts if p.strip()]


def main():
    tables = parse_tables("3, 11, 12")
    print(tables)
    print(", ".join(str(t) for t in tables))
    print("3,,12".split(","))
    print("  3   11 12 ".split())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python split_tables.py
```

Output:

```
[3, 11, 12]
3, 11, 12
['3', '', '12']
['3', '11', '12']
```

`join` needs strings, so `str(t)` on each int. An empty field from `split(",")` is why `if p.strip()` is there. `int("")` would raise `ValueError`.

Case 3: f-strings at the desk

```
# ticket_fstring.py
def main():
    ticket = {"id": 7, "table": 12, "cents": 1850, "note": "window"}
    print(f"ticket {ticket['id']} → table {ticket['table']}")
    print(f"${ticket['cents'] / 100:.2f}")
    print(f"note={ticket['note']!r}")
    width = 4
    print(f"table {ticket['table']:>{width}}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_fstring.py
```

Output:

```
ticket 7 → table 12
$18.50
note='window'
table   12
```

`!r` is what you want in logs so spaces and empty strings stay visible. Nested quotes: use the other quote around the f-string, or index with a name first (`tid = ticket["id"]`).

Case 4: Encode at the edge

```
# ticket_encode.py
def main():
    text = "café 12"
    raw = text.encode("utf-8")
    print(raw)
    print(raw.decode("utf-8"))
    try:
        raw.decode("ascii")
    except UnicodeDecodeError as e:
        print(type(e).__name__)
    print(text.encode("ascii", errors="replace"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_encode.py
```

Output:

```
b'caf\xc3\xa9 12'
café 12
UnicodeDecodeError
b'caf? 12'
```

UTF-8 round-trips. ASCII cannot hold `é`, so `"strict"` raises and `"replace"` writes `?`. Do not use `"ignore"` for data you care about — it drops bytes with no trace.

The trap

Treating `bytes` as a string, or building a path / JSON body as bytes by accident.

```
# join_bytes.py
def main():
    tables = [3, 11]
    line = ",".join(tables)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python join_bytes.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "join_bytes.py", line 8, in <module>
    main()
  File "join_bytes.py", line 4, in main
    line = ",".join(tables)
TypeError: sequence item 0: expected str instance, int found
```

`join` does not stringify for you. The bytes twin is `b",".join([b"3", b"11"])` — still bytes, still not mixed with `str`. Convert first: `",".join(str(t) for t in tables)`.

The boring rule

- `str` in memory. `bytes` on the wire. UTF-8 at the door.
- Assign the result of `replace`, `strip`, `upper`, `encode`.
- `split` to parse. `join` to render. The joiner is the separator.
- f-strings for messages. `!r` in logs.
- Never `+` a `str` and `bytes`. Never `join` ints.

Try this

1. In `rewrite_label.py`, `strip` a label with spaces (`"  table 3  "`) and assign it back.
2. Extend `parse_tables` to reject a token that is not all digits (`"3, eleven"`) with `ValueError`.
3. In `ticket_encode.py`, encode with `"utf-16"` and print `len(raw)` versus `len(text)`.

Dicts

Dicts

A **dict** maps keys to values. The boring default at the desk is a dict for a ticket that is still growing fields, `d["id"]` when the key must exist, `d.get("note", "")` when it might not, and `|` to merge overlays without mutating the originals.

Mental model

Keys must be hashable (`int`, `str`, `tuple` of hashables, `frozenset` — not `list`, not `dict`). Values can be anything. Lookup is by key, not by position.

Since Python 3.7, **insertion order is part of the language**. `for key in d:` walks keys in the order they were first inserted. Updating a key in place does not move it. Deleting and re-inserting puts it at the end.

`d[k]` raises `KeyError` if `k` is missing. `d.get(k)` returns `None`. `d.get(k, default)` returns the default. `d.setdefault(k, [])` inserts the default if missing and returns the value — handy, and easy to mutate a shared list if you are careless.

`a | b` is a new dict: keys from `a`, then keys from `b` overwrite. `a |= b` updates `a` in place. `{**a, **b}` is the older unpack form. Prefer `|` when both sides are dicts.

Worked examples

Case 1: Keys, order, update in place

Save as `ticket_dict.py`.

```
# ticket_dict.py
def main():
    ticket = {"id": 7, "table": 12}
    ticket["status"] = "open"
    ticket["table"] = 3
    print(list(ticket))
    print(list(ticket.values()))
    for key, value in ticket.items():
        print(key, value)


if __name__ == "__main__":
```

```
    main()
```

Run:

```
uv run python ticket_dict.py
```

Output:

```
['id', 'table', 'status']
[7, 3, 'open']
id 7
table 3
status open
```

`table` stayed in its original position after the update. `status` was appended. `list(ticket)` is the keys.

Case 2: `get` when a field is optional

```
# ticket_note.py
def note_line(ticket):
    note = ticket.get("note", "")
    if note:
        return f"ticket {ticket['id']}: {note}"
    return f"ticket {ticket['id']}: (no note)"


def main():
    print(note_line({"id": 7, "note": "window"}))
    print(note_line({"id": 8}))
    print({"id": 8}.get("note"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_note.py
```

Output:

```
ticket 7: window
ticket 8: (no note)
None
```

`get("note")` without a default is `None`, which is easy to confuse with a stored `None`. Prefer an explicit default when the rest of the function wants a string.

Case 3: Merge with |

```
# merge_ticket.py
def main():
    defaults = {"status": "open", "cents": 0, "table": 0}
    incoming = {"id": 7, "table": 12, "cents": 1850}
    ticket = defaults | incoming
    print(ticket)
    override = {"status": "paid"}
    print(ticket | override)
    print(ticket)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python merge_ticket.py
```

Output:

```
{'status': 'open', 'cents': 1850, 'table': 12, 'id': 7}
{'status': 'paid', 'cents': 1850, 'table': 12, 'id': 7}
{'status': 'open', 'cents': 1850, 'table': 12, 'id': 7}
```

`defaults | incoming` keeps default key order, then adds new keys (`id`) at the end. `incoming` wins on `cents` and `table`. The last `print` shows `|` did not mutate `ticket`.

The trap

`KeyError` on a key you assumed was there. It is the right exception when the ticket is corrupt. It is a trap when the field is optional and you meant `get`.

```
# missing_note.py
def print_note(ticket):
    print(ticket["note"])


def main():
    print_note({"id": 8, "table": 3})
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python missing_note.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "missing_note.py", line 11, in <module>
    main()
  File "missing_note.py", line 7, in main
    print_note({"id": 8, "table": 3})
  File "missing_note.py", line 3, in print_note
    print(ticket["note"])
KeyError: 'note'
```

Fix for optional: `ticket.get("note", "")`. Fix for required: let `KeyError` surface, or raise `ValueError("ticket missing note")` with the ticket id. Do not write `if "note" in ticket` and then silently skip a required field.

A second trap: `d[k] += 1` also raises `KeyError` on a missing key. Use `d[k] = d.get(k, 0) + 1` or `collections.Counter`.

The boring rule

- Required field: `d[k]`. Optional field: `d.get(k, default)`.
- Merge overlays with `a | b`. Do not mutate defaults.
- Trust insertion order. Do not sort keys unless the output needs a sort.
- Keys are unique. A second write overwrites.
- `KeyError` means the shape is wrong. Do not catch it to return `None` unless that is the API.

Try this

1. In `ticket_dict.py`, `del ticket["table"]` then set `ticket["table"] = 12` again. Print `list(ticket)` and notice `table` moved to the end.
2. Merge three dicts: `defaults | incoming | {"status": "fired"}`.
3. Change `print_note` to use `get` and print `(no note)` when missing. Re-run `missing_note.py`.

Sets

Sets

A **set** holds unique hashable values. The boring default is a set when the question is **membership** or **overlap** (which tables are free, which allergens collided), not when order or duplicates matter.

Mental model

`{3, 11, 12}` is a set. `set()` is the empty set — `{}` is an empty **dict**. Values must be hashable. A set itself is mutable and not hashable. **frozenset** is the immutable twin; it can be a dict key or a member of another set.

Operators:

Op	Meaning
`a | b`	union — in either
`a & b`	intersection — in both
`a - b`	difference — in `a` not `b`
`a ^ b`	symmetric difference — in one, not both
`x in a`	membership

`<=` / `<` are subset. `>=` / `>` are superset. Methods (`union`, `intersection`, `add`, `discard`, `remove`) exist too. `remove` raises `KeyError` if missing; `discard` does not.

Iteration order is not a contract. Sort when you print.

Worked examples

Case 1: Unique tables on the shift

Save as `unique_tables.py`.

```
# unique_tables.py
def main():
    landed = [12, 3, 12, 11, 3]
    tables = set(landed)
    print(sorted(tables))
    print(12 in tables)
    tables.add(4)
    tables.discard(99)
```

```
    print(sorted(tables))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python unique_tables.py
```

Output:

```
[3, 11, 12]
True
[3, 4, 11, 12]
```

`set(landed)` dropped duplicate `12` and `3`. `discard(99)` is a no-op. `remove(99)` would raise.

Case 2: Operators for two shifts

```
# shift_overlap.py
def main():
    lunch = {3, 11, 12}
    dinner = {12, 14, 15}
    print("either ", sorted(lunch | dinner))
    print("both   ", sorted(lunch & dinner))
    print("lunch only", sorted(lunch - dinner))
    print("one side", sorted(lunch ^ dinner))
    print("lunch subset of either", lunch <= (lunch | dinner))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_overlap.py
```

Output:

```
either  [3, 11, 12, 14, 15]
both    [12]
lunch only [3, 11]
one side [3, 11, 14, 15]
lunch subset of either True
```

Read `&` as “conflict” when both shifts claimed the same table. Read `-` as “only lunch.”

Case 3: Allergens as a set, frozenset as a key

```
# allergens.py
def main():
    soup = frozenset({"dairy", "gluten"})
    tea = frozenset()
    cake = frozenset({"dairy", "nuts"})
    by_item = {
        soup: "soup",
        tea: "tea",
        cake: "cake",
    }
    guest = {"dairy"}
    for tags, name in by_item.items():
        hit = set(tags) & guest
        if hit:
            print(name, "hits", sorted(hit))
        else:
            print(name, "ok")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python allergens.py
```

Output:

```
soup hits ['dairy']
tea ok
cake hits ['dairy']
```

A plain `set` cannot be a dict key (`TypeError: cannot use 'set' as a dict key`). `frozenset` can. `frozenset()` is the empty tag set for tea. Converting with `set(tags)` lets you `&` against the mutable guest set.

The trap

Building a set of dicts (tickets), or expecting a set to remember insert order in your output.

```
# set_of_tickets.py
def main():
    tickets = [{"id": 7}, {"id": 8}]
    print(set(tickets))
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python set_of_tickets.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "set_of_tickets.py", line 8, in <module>
    main()
  File "set_of_tickets.py", line 4, in main
    print(set(tickets))
TypeError: cannot use 'dict' as a set element (unhashable type: 'dict')
```

Sets hash their members. Dicts (and lists) do not hash. Store ids: `set(t["id"] for t in tickets)`. If you need unique tickets as whole records, pick a key (`id`) or use a dict keyed by id.

`{1, 2, 3}` looks ordered in a lucky print. Do not write tests that `==` a printed order. `sorted(the_set)` is the boring print.

The boring rule

- Set: unique membership. List: sequence. Dict: named fields.
- Use `| & - ^` and `in`. Sort for display.
- `discard` when missing is fine. `remove` when missing is a bug.
- `frozenset` for keys and for set-of-sets.
- Empty set is `set()`, never `{}`.

Try this

1. In `unique_tables.py`, call `tables.remove(99)` instead of `discard` and read the `KeyError`.
2. Add a `brunch` set and print tables that appear in all three shifts (`lunch & dinner & brunch`).
3. Change `allergens.py` so `guest = {"nuts"}` and confirm only cake hits.

Dataclasses and Data Modeling

Dataclasses and Data Modeling

A **dataclass** is a class whose job is to hold fields. The boring default for a ticket that has a stable shape is `@dataclass` with named fields, not a dict you poke with string keys, and not a hand-written class with ten identical `__init__` lines.

Field names on a dataclass need a type (the decorator uses the annotations to know what the fields are). That is the exception in this part of the book. Functions still have no type hints. Full typing comes later.

Mental model

`@dataclass` generates `__init__`, `__repr__`, and `__eq__` from the field list. Two tickets with the same field values compare equal. The repr prints the class name and fields, which is what you want in a failure.

Defaults go after fields without defaults. Mutable defaults still need `field(default_factory=list)` — the same trap as function arguments, with the same fix.

`replace(ticket, status="paid")` returns a **new** instance. `asdict(ticket)` is a dict for JSON-shaped edges. Neither is magic: nested dataclasses become nested dicts only if you ask (`asdict`).

`frozen=True` makes assignment to fields raise. Use it when the ticket should not change in place (good for dict keys if all fields are hashable). The default is mutable, which matches a ticket that moves from `"open"` to `"paid"`.

A dataclass is still a class. Put tiny methods on it when the behavior belongs to the ticket (`is_open`, `label`). Keep I/O and kitchen routing as functions.

Worked examples

Case 1: A ticket with fields

Save as `ticket_dc.py`.

```
# ticket_dc.py
from dataclasses import dataclass


@dataclass
class Ticket:
```

```
    id: int
    table: int
    status: str = "open"
    cents: int = 0


def main():
    t = Ticket(7, 12, cents=1850)
    print(t)
    print(t.id, t.table, t.status)
    t.status = "paid"
    print(t)
    print(Ticket(7, 12, "open", 1850) == Ticket(7, 12, "open", 1850))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_dc.py
```

Output:

```
Ticket(id=7, table=12, status='open', cents=1850)
7 12 open
Ticket(id=7, table=12, status='paid', cents=1850)
True
```

`Ticket(7, 12, cents=1850)` uses the default `status`. Equality is by value, not by identity. Two separately constructed tickets with the same numbers are equal.

Case 2: Methods that belong to the ticket

```
# ticket_label.py
from dataclasses import dataclass


@dataclass
class Ticket:
    id: int
    table: int
    status: str
    cents: int

    def label(self):
```

```
        return f"ticket {self.id} → table {self.table}"

    def is_open(self):
        return self.status == "open"


def main():
    t = Ticket(7, 12, "open", 1850)
    print(t.label())
    print(t.is_open())
    t.status = "paid"
    print(t.is_open())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_label.py
```

Output:

```
ticket 7 → table 12
True
False
```

`label` and `is_open` read the fields. They do not print, they do not hit the network. That is the size of method you want on a dataclass.

Case 3: `replace`, `asdict`, and a list factory

```
# ticket_replace.py
from dataclasses import asdict, dataclass, field, replace


@dataclass
class Ticket:
    id: int
    table: int
    status: str = "open"
    notes: list = field(default_factory=list)


def main():
    t = Ticket(7, 12)
```

```
    t.notes.append("window")
    paid = replace(t, status="paid")
    print(t)
    print(paid)
    print(asdict(paid))
    u = Ticket(8, 3)
    print(u.notes)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_replace.py
```

Output:

```
Ticket(id=7, table=12, status='open', notes=['window'])
Ticket(id=7, table=12, status='paid', notes=['window'])
{'id': 7, 'table': 12, 'status': 'paid', 'notes': ['window']}
[]
```

`replace` copied the notes **list object**. `paid.notes is t.notes` is `True`. If notes must fork, pass a new list into `replace`. `default_factory=list` gives ticket 8 its own empty list — not the sticky shared list from a `notes: list = []` default.

The trap

A mutable default on a field is the same sticky-list bug as a default argument. Dataclasses refuse it at class definition.

```
# sticky_field.py
from dataclasses import dataclass


@dataclass
class Ticket:
    id: int
    notes: list = []


def main():
    print(Ticket(7))
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python sticky_field.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "sticky_field.py", line 5, in <module>
    @dataclass
  File ".../dataclasses.py", line 917, in _get_field
    raise ValueError(...)
ValueError: mutable default <class 'list'> for field notes is not allowed: use default_factory
```

The decorator is doing you a favor. The fix is Case 3: `notes: list = field(default_factory=list)`.

`replace` still shares nested lists: `paid.notes is t.notes`. Copy the list when the new ticket must own its notes.

The boring rule

- Stable shape → dataclass. Ad-hoc bag → dict, then promote.
- Annotate fields (`id: int`). Do not decorate every function yet.
- `field(default_factory=list)` for mutable defaults.
- Tiny methods for labels and predicates. Functions for workflows.
- `replace` for a changed copy. Remember nested lists are still aliases.
- `frozen=True` when the record should not move in place.

Try this

1. Add a `covers: int = 0` field to `Ticket` in `ticket_dc.py` and print it.
2. In `ticket_replace.py`, `replace(t, notes=list(t.notes))` then append `"allergy"` only on `paid`. Confirm `t.notes` did not gain it.
3. Rebuild `Ticket` with `@dataclass(frozen=True)` in a copy of `ticket_dc.py`. Assign `t.status = "paid"` and read the exception.

Functions

Functions in Depth

Functions in Depth

A **function** is a named piece of work with a clear input and a clear output. The boring default is a short `def`, an explicit `return`, and arguments you can read at the call site. Fancy packing (`*args`, `**kwargs`) is for the edges, not for every helper.

Mental model

`def` binds a name to a function object. **Parameters** are the names in the `def` line. **Arguments** are the values you pass at the call. If you omit `return`, Python returns `None`.

Several values come back as a **tuple**. The caller unpacks them, or keeps the tuple.

A **default** fills in a parameter you skip. A **keyword-only** parameter (after a bare `*`) must be passed by name, so `tax=0.1` cannot be confused with another number. `*args` gathers extra positional arguments into a tuple. `**kwargs` gathers extra keywords into a dict. A leading `*` or `**` at the call site **unpacks** a sequence or a dict into arguments.

Worked examples

Case 1: `def` and `return`

Save as `ticket_label.py`. The function takes a ticket dict and returns a string. `print` happens in `main`, not inside `label`.

```
# ticket_label.py
def label(ticket):
    return f"ticket {ticket['id']} → table {ticket['table']}"


def main():
    t = {"id": 7, "table": 12}
    text = label(t)
    print(text)
    print(label.__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_label.py
```

Output:

```
ticket 7 → table 12
label
```

The function object has a `__name__`. You will need that later for decorators.

Case 2: Several values as a tuple

Save as `split_shift.py`. A shift string `"09:00-17:00"` splits into start and end. Returning two strings is returning one tuple.

```
# split_shift.py
def split_shift(text):
    start, end = text.split("-", 1)
    return start, end


def main():
    pair = split_shift("09:00-17:00")
    print(type(pair).__name__, pair)
    start, end = split_shift("09:00-17:00")
    print(start)
    print(end)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python split_shift.py
```

Output:

```
tuple ('09:00', '17:00')
09:00
17:00
```

Name the two results at the call site. Do not invent a one-off class for two strings.

Case 3: Defaults

Save as `add_tax.py`. Most orders use the desk rate. A catering order can override it.

```
# add_tax.py
def with_tax(cents, rate=0.1):
    return round(cents * (1 + rate))


def main():
    print(with_tax(400))
    print(with_tax(400, 0.0))
    print(with_tax(400, rate=0.2))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python add_tax.py
```

Output:

```
440
400
480
```

Defaults are evaluated **once**, when `def` runs, not on every call. That fact is the trap at the end of this chapter.

Case 4: Keyword-only arguments

Save as `charge.py`. After the `*`, `tax` and `tip` must be passed by name. A second positional number is a `TypeError`, not a silent mix-up of tax and tip.

```
# charge.py
def charge(cents, *, tax=0.1, tip=0):
    return round(cents * (1 + tax) + tip)


def main():
    print(charge(400))
    print(charge(400, tax=0.2, tip=50))
    try:
        charge(400, 0.2)
    except TypeError as e:
```

```
        print(type(e).__name__ + ":", e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python charge.py
```

Output:

```
440
530
TypeError: charge() takes 1 positional argument but 2 were given
```

Use keyword-only parameters when two numbers would be easy to swap.

Case 5: `*args`, `**kwargs`, and unpacking

Save as `announce.py`. Extra words land in `parts`. Extra flags land in `flags`. At the call site, `*` unpacks a tuple and `**` unpacks a dict.

```
# announce.py
def announce(*parts, **flags):
    text = " ".join(str(p) for p in parts)
    if flags.get("shout"):
        text = text.upper()
    return text


def main():
    row = ("ticket", 7, "table", 12)
    print(announce(*row))
    opts = {"shout": True}
    print(announce("desk", "open", **opts))
    print(announce("quiet", "please"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python announce.py
```

Output:

```
ticket 7 table 12
DESK OPEN
quiet please
```

Reach for `*args` / `**kwargs` when you are forwarding arguments to another function. Do not use them to avoid naming the three parameters you actually have.

The trap

A default list is **one** list, shared by every call that omits `bag`. The second `add_item("tea")` still sees the latte.

Save as `shared_bag.py`:

```
# shared_bag.py
def add_item(item, bag=[]):
    bag.append(item)
    return bag


def add_item_ok(item, bag=None):
    if bag is None:
        bag = []
    bag.append(item)
    return bag


def main():
    a = add_item("latte")
    b = add_item("tea")
    print("shared:", a, b, a is b)
    c = add_item_ok("latte")
    d = add_item_ok("tea")
    print("fresh:", c, d, c is d)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shared_bag.py
```

Output:

```
shared: ['latte', 'tea'] ['latte', 'tea'] True
fresh: ['latte'] ['tea'] False
```

The boring default for a mutable default is `None`, then a new list (or dict, or set) inside the function.

The boring rule

- Return a value. Leave `print` to the caller unless the function *is* the printer.
- Several results: return a tuple and unpack it.
- Defaults for immutable values (`int`, `str`, `None`, `False`) are fine. Mutable defaults are not.
- Make easy-to-swap numbers keyword-only.
- Name parameters. Use `*args` / `**kwargs` to forward, not to hide.
- Unpack with `*` and `**` at the call site when you already have a sequence or a dict.

Try this

1. In `ticket_label.py`, return `None` on purpose (no `return`) and print the result. Then put the `return` back.
2. In `split_shift.py`, add a third value: the length of the shift in hours, as a string like `"8h"`. Unpack three names.
3. In `charge.py`, add keyword-only `service=0`. Show that `charge(400, 1)` still fails and `charge(400, service=1)` works.
4. In `shared_bag.py`, pass an explicit list into `add_item_ok` twice and confirm it *does* grow — that is the caller's list, not a hidden default.

Functions as Values

Functions as Values

A function is a value. You can pass it, store it, and return it. The boring default is a named `def` you hand to `sorted`, `min`, or a small dispatch dict. Keep `lambda` tiny. Prefer a comprehension over `map` and `filter`.

Mental model

A **callable** is anything you can follow with `(...)`. Function objects are callables. So are methods. `sorted` does not know about tickets; it only knows how to compare keys. You pass `key=some_function` and `sorted` calls that function once per item.

A **dispatch table** is a dict from a name to a function. Instead of a long `if/elif` chain that you forget to extend, you look up the function and call it.

A **lambda** is a one-expression function with no name. If it needs a statement, a name, or more than one expression, use `def`.

`map` and `filter` apply a function to an iterable and return an iterator. A **list comprehension** does the same job with ordinary `for` and `if`. That is the form a teammate will edit at 5 p.m.

Worked examples

Case 1: Pass a function to `sorted`

Save as `sort_tickets.py`. `by_table` is an ordinary function. `sorted` calls it for each ticket.

```
# sort_tickets.py
def by_table(ticket):
    return ticket["table"]


def main():
    tickets = [
        {"id": 9, "table": 12},
        {"id": 3, "table": 4},
        {"id": 7, "table": 12},
    ]
    for t in sorted(tickets, key=by_table):
        print(f"ticket {t['id']} table {t['table']}")
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python sort_tickets.py
```

Output:

```
ticket 3 table 4
ticket 9 table 12
ticket 7 table 12
```

Equal tables keep their original order. `sorted` is stable.

Case 2: A dict of functions

Save as `dispatch.py`. The desk has three verbs. Each verb is a function. Unknown verbs raise `KeyError`, which you turn into a `ValueError` with a clear message.

```
# dispatch.py
def open_ticket(n):
    return f"opened table {n}"


def pay_ticket(n):
    return f"paid table {n}"


def close_ticket(n):
    return f"closed table {n}"


HANDLERS = {
    "open": open_ticket,
    "pay": pay_ticket,
    "close": close_ticket,
}


def run(verb, n):
    try:
        fn = HANDLERS[verb]
    except KeyError:
        raise ValueError(f"unknown verb {verb!r}") from None
```

```
    return fn(n)


def main():
    print(run("open", 12))
    print(run("pay", 12))
    try:
        run("refund", 12)
    except ValueError as e:
        print(type(e).__name__ + ":", e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python dispatch.py
```

Output:

```
opened table 12
paid table 12
ValueError: unknown verb 'refund'
```

Add a verb by writing a function and one dict line. Do not grow a twelve-branch `if`.

Case 3: A tiny `lambda`

Save as `tiny_lambda.py`. The key is one attribute lookup. A lambda is acceptable here. The next line uses a named function for the same job so you can see both.

```
# tiny_lambda.py
def by_id(ticket):
    return ticket["id"]


def main():
    tickets = [
        {"id": 9, "table": 12},
        {"id": 3, "table": 4},
    ]
    by_lambda = sorted(tickets, key=lambda t: t["id"])
    by_def = sorted(tickets, key=by_id)
    print([t["id"] for t in by_lambda])
    print([t["id"] for t in by_def])
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python tiny_lambda.py
```

Output:

```
[3, 9]
[3, 9]
```

If the key grows a second line, delete the lambda and keep the `def`.

Case 4: `map` / `filter` versus a comprehension

Save as `open_tables.py`. Both styles produce the same list. The comprehension is the one you keep.

```
# open_tables.py
def main():
    tickets = [
        {"id": 9, "table": 12, "status": "open"},
        {"id": 3, "table": 4, "status": "paid"},
        {"id": 7, "table": 12, "status": "open"},
    ]
    mapped = list(map(lambda t: t["id"], tickets))
    filtered = list(filter(lambda t: t["status"] == "open", tickets))
    ids = [t["id"] for t in tickets]
    open_ids = [t["id"] for t in tickets if t["status"] == "open"]
    print("map:", mapped)
    print("filter ids:", [t["id"] for t in filtered])
    print("comp:", ids)
    print("comp open:", open_ids)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python open_tables.py
```

Output:

```
map: [9, 3, 7]
filter ids: [9, 7]
comp: [9, 3, 7]
comp open: [9, 7]
```

`map` and `filter` are not wrong. They are just noisier once the function is a lambda. A comprehension already has `for` and `if`.

The trap

Assigning a lambda to a name is a `def` with worse traceback names and no statements. `ruff` will flag it (`E731`). Use `def`.

Save as `named_lambda.py`:

```
# named_lambda.py
def main():
    bump = lambda cents: cents + 50
    print(bump(400))
    print(bump.__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python named_lambda.py
```

Output:

```
450
<lambda>
```

The fix is a one-line `def`:

```
# named_def.py
def bump(cents):
    return cents + 50


def main():
    print(bump(400))
    print(bump.__name__)
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python named_def.py
```

Output:

```
450
bump
```

Same work. A real name in traces and in `help`.

The boring rule

- Pass named functions into `sorted`, `min`, `max`, and your own helpers.
- A small dict of functions is better than a growing `if`/`elif` ladder.
- `lambda` is for a single expression you will not reuse. Do not assign it to a name.
- Prefer `[... for ... in ... if ...]` over `list(map(...))` and `list(filter(...))`.
- If the callable needs a body, it needs a `def`.

Try this

1. In `sort_tickets.py`, sort by `(table, id)` so table 12's tickets come out 7 then 9. A named function that returns a tuple is fine.
2. In `dispatch.py`, add `void` that returns `"voided table {n}"` and call it.
3. In `open_tables.py`, drop `map` and `filter`. Keep only the comprehensions. Print tickets whose table is 12.
4. Rewrite `named_lambda.py` so `bump` also prints nothing extra — just return the value — using `def`.

Decorators

Decorators

A **decorator** is a function that takes a function and returns a replacement. The boring default is a thin wrapper that logs, times, or checks something, then calls the original. Always apply `functools.wraps` so the wrapper keeps the original name and docstring.

Mental model

```
@log
def open_table(n):
    ...
```

means "define `open_table`, then set `open_table = log(open_table)`." The replacement is what callers invoke.

The wrapper should accept `*args, **kwargs` unless you know the exact signature. It should return whatever the original returns (or raise whatever it raises).

`functools.wraps(fn)` copies `fn`'s `__name__`, `__doc__`, and a few other attributes onto the wrapper. Without it, traces and `help` talk about `wrapper`.

Stacked decorators apply **bottom first**: `@a` then `@b` on `f` is `f = a(b(f))`.

Worked examples

Case 1: A function that wraps a function

Save as `log_open.py`. `log` prints the name and the arguments, then calls the original.

```
# log_open.py
def log(fn):
    def wrapper(*args, **kwargs):
        print(f"call {fn.__name__}{args}")
        return fn(*args, **kwargs)

    return wrapper


@log
def open_table(n):
```

```
    return f"opened table {n}"


def main():
    print(open_table(12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python log_open.py
```

Output:

```
call open_table(12,)
opened table 12
```

That is the whole trick. No metaclass. No import magic.

Case 2: `functools.wraps`

Save as `wrapped_open.py`. After `@wraps(fn)`, the public name is still `open_table`.

```
# wrapped_open.py
from functools import wraps


def log(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f"call {fn.__name__}")
        return fn(*args, **kwargs)

    return wrapper


@log
def open_table(n):
    """Open one table and return a label."""
    return f"opened table {n}"


def main():
    print(open_table.__name__)
```

```
    print(open_table.__doc__)
    print(open_table(4))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python wrapped_open.py
```

Output:

```
open_table
Open one table and return a label.
call open_table
opened table 4
```

Keep `@wraps` even on internal tools. Tests and traces will thank you.

Case 3: Stacked decorators

Save as `stacked.py`. `count_calls` runs first (closer to the function). `log` wraps that result. Both use `wraps`.

```
# stacked.py
from functools import wraps


def log(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        print(f"log {fn.__name__}")
        return fn(*args, **kwargs)

    return wrapper


def count_calls(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        print(f"count {wrapper.calls}")
        return fn(*args, **kwargs)

    wrapper.calls = 0
```

```
    return wrapper


@log
@count_calls
def ping():
    return "pong"


def main():
    print(ping())
    print(ping())
    print("name", ping.__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python stacked.py
```

Output:

```
log ping
count 1
pong
log ping
count 2
pong
name ping
```

Read the source from the bottom: `ping` is counted, then logged. Swap the two `@` lines if you want the other order, and print again.

The trap

Without `wraps`, the object people import is named `wrapper`. `help`, traces, and a registry of `__name__` all lie.

Save as `bare_wrapper.py`:

```
# bare_wrapper.py
def log(fn):
    def wrapper(*args, **kwargs):
        return fn(*args, **kwargs)
```

```
    return wrapper


@log
def open_table(n):
    """Open one table and return a label."""
    return f"opened table {n}"


def main():
    print(open_table.__name__)
    print(open_table.__doc__)
    print(open_table(12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python bare_wrapper.py
```

Output:

```
wrapper
None
opened table 12
```

The return value still works. The metadata does not. Copy the decorator from `wrapped_open.py` — `@wraps(fn)` on the inner function — and `__name__` becomes `open_table` again.

The boring rule

- A decorator is `def deco(fn):` plus an inner `wrapper` plus `return wrapper`.
- Put `@wraps(fn)` on every wrapper you ship.
- Forward with `*args, **kwargs` and return the original result.
- Stacked `@` lines: bottom decorator runs first.
- Do not decorate a function to look clever. Decorate when many functions share one extra behaviour (log, retry, require a role).

Try this

1. In `log_open.py`, make `open_table` take `n` and `label`, and log both. Call `open_table(12, label="patio")`.

2. In `wrapped_open.py`, print `open_table.__wrapped__(12)` and confirm it skips the log line.
3. In `stacked.py`, swap `@log` and `@count_calls`. Run it. Note which line prints first.
4. Fix `bare_wrapper.py` with `functools.wraps` so `__name__` is `open_table` and `__doc__` is the docstring.

Closures and Scope

Closures and Scope

A nested function can remember names from the function that created it. That memory is a **closure**. The boring default is a small factory: `make_taxer(rate)` returns a function that always uses that rate. The trap is a loop: every nested function sees the **last** value of the loop variable unless you bind it as a default argument.

Mental model

Python looks up a name in this order: local, enclosing function, global (module), then built-ins. That is **LEGB**. A nested `def` that reads a name from the enclosing function is a closure. The inner function stores a cell pointing at that name, not a snapshot of the value at `def` time.

Late binding means the value is read when the inner function *runs*, not when it was defined. In a loop, that name is the last assignment.

The usual fix is a default argument: `def inner(n=n):`. Defaults are evaluated at `def` time, so each inner function gets its own copy.

`nonlocal` lets the inner function **assign** to a name in the enclosing function. Use it for a tiny counter. Do not use it as a hidden global.

Worked examples

Case 1: A factory that closes over a rate

Save as `make_taxer.py`. Each returned function remembers its own `rate`.

```
# make_taxer.py
def make_taxer(rate):
    def with_tax(cents):
        return round(cents * (1 + rate))

    return with_tax


def main():
    desk = make_taxer(0.1)
    catering = make_taxer(0.0)
    print(desk(400))
```

```
    print(catering(400))
    print(desk.__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python make_taxer.py
```

Output:

```
440
400
with_tax
```

`desk` and `catering` are two functions. They share code, not data.

Case 2: `nonlocal` for a running count

Save as `shift_counter.py`. `bump` assigns to `n` in `make_counter`. Without `nonlocal`, that assignment would create a new local `n` and the outer `n` would never change.

```
# shift_counter.py
def make_counter():
    n = 0

    def bump():
        nonlocal n
        n += 1
        return n

    return bump


def main():
    tickets = make_counter()
    print(tickets())
    print(tickets())
    other = make_counter()
    print(other())
    print(tickets())


if __name__ == "__main__":
```

```
    main()
```

Run:

```
uv run python shift_counter.py
```

Output:

```
1
2
1
3
```

Two counters do not share `n`. That is the point of a closure: state that is not a global and not a class — yet.

If you need to inspect the count, name it on an object. A closure with `nonlocal` is easy to overgrow.

Case 3: Late binding in a loop (the bug, then the default-arg fix)

Save as `late_tables.py`. `make_buggy` appends three functions in a loop. Each function reads `n`. When they run, the loop is finished, so `n` is `12` for all three.

`make_fixed` binds `n` as a default: `def open_it(n=n)`. Defaults freeze the value at `def` time. Each function keeps the table number from that iteration.

```
# late_tables.py
def make_buggy(tables):
    fns = []
    for n in tables:
        def open_it():
            return f"open table {n}"

        fns.append(open_it)
    return fns


def make_fixed(tables):
    fns = []
    for n in tables:
        def open_it(n=n):
            return f"open table {n}"

        fns.append(open_it)
    return fns
```

```
def main():
    tables = [3, 7, 12]
    print("buggy:")
    for fn in make_buggy(tables):
        print(fn())
    print("fixed:")
    for fn in make_fixed(tables):
        print(fn())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python late_tables.py
```

Output:

```
buggy:
open table 12
open table 12
open table 12
fixed:
open table 3
open table 7
open table 12
```

The inner parameter `n` shadows the loop variable. That is deliberate. Callers still write `fn()` with no arguments; they never pass `n`.

A second honest fix is a factory: `def make_open(n): def open_it(): return ...; return open_it` and append `make_open(n)`. Same idea: bind `n` as a parameter of a new call. The default-arg form is the one-liner you will see in reviews.

The trap

The trap **is** Case 3. People write the buggy loop because it looks like the function captured `3`, then `7`, then `12`. It captured the **name** `n`. Read `late_tables.py` from the top. If you only remember one trick from this chapter, remember `n=n`.

A cousin of the same bug: a list of lambdas, `lambda: n`, inside a `for`. Same late binding. Same default-arg fix: `lambda n=n: n`.

The boring rule

- Use a closure as a tiny factory (`make_taxer(rate)`), not as a hidden object model.
- Lookup is LEGB. Assignment is local unless you write `nonlocal` or `global`.
- Prefer `nonlocal` only for a small counter or flag. A class is clearer once you have two pieces of state.
- Never build functions in a loop that close over the loop variable without binding it (`n=n` or a factory).
- Do not use `global` to pass data into a function. Pass an argument.

Try this

1. In `make_taxer.py`, add `make_discounter(percent)` that returns a function subtracting that percent from cents. Print `make_discounter(10)(400)`.
2. In `shift_counter.py`, add a nested `reset` that sets `n` back to 0 with `nonlocal`. Call `bump`, `reset`, `bump`.
3. In `late_tables.py`, add `make_lambda(tables)` that uses `lambda n=n: f"open table {n}"`. Confirm it matches `make_fixed`.
4. Comment out `n=n` in `make_fixed` (empty `def open_it():`) and run it. You should see the buggy output again.

Modules

Modules and Imports

Modules and Imports

A **module** is a `.py` file. `import` runs that file once and binds a name to the module object. The boring default is `import prices` (or `from prices import cents` when one name is the whole API). Keep a folder of files, not a pile of copy-paste.

Mental model

`import prices` looks for `prices.py` (or a package named `prices`) on `sys.path`. The first time, Python executes the file, stores the module in `sys.modules`, and binds the name `prices` in your module. The second `import` reuses `sys.modules`. It does not run the file again.

`from prices import cents` still loads the whole module. It then binds `cents` in *your* namespace. The name `prices` is not bound unless you also imported it.

`import prices as p` binds a shorter name. Use `as` when two modules share a last component, or when the real name is awkward.

`importlib.import_module("prices")` does the same lookup from a **string**. That is how you load a plugin name from config. You still need a real module on the path.

Worked examples

Put every file in this chapter in the **same folder**. Run the commands from that folder.

Case 1: `import` a sibling file

Save as `prices.py`:

```
# prices.py
MENU = {"latte": 400, "tea": 250, "bun": 150}


def cents(name):
    if name not in MENU:
        raise KeyError(f"unknown item {name!r}")
    return MENU[name]
```

Save as `order.py`. This is the program you run.

```
# order.py
import prices


def total(names):
    return sum(prices.cents(n) for n in names)


def main():
    print(prices.cents("latte"))
    print(total(["latte", "tea"]))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python order.py
```

Output:

```
400
650
```

`prices.py` has no `main` guard. It is a library file. `order.py` is the program.

Case 2: `from` import

Save as `from_order.py`. Only `cents` is bound here. `MENU` is still on the module, but this file cannot see the name `prices`.

```
# from_order.py
from prices import cents


def main():
    print(cents("bun"))
    print(cents.__module__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python from_order.py
```

Output:

```
150
prices
```

`cents.__module__` still says `prices`. You imported the function, not a copy of its code.

Case 3: `import ... as`

Save as `alias_order.py`. The alias is local to this file.

```
# alias_order.py
import prices as menu


def main():
    print(menu.cents("tea"))
    print(menu.MENU["tea"])


if __name__ == "__main__":
    main()
```

Run:

```
uv run python alias_order.py
```

Output:

```
250
250
```

Case 4: `importlib` from a string

Save as `load_prices.py`. The module name comes from a variable. That is the only reason to use `importlib` on day one.

```
# load_prices.py
import importlib


def main():
    name = "prices"
```

```
    mod = importlib.import_module(name)
    print(mod.cents("latte"))
    print(mod.__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python load_prices.py
```

Output:

```
400
prices
```

If `name` is wrong, you get `ModuleNotFoundError`. Do not catch that and keep going unless you are probing optional plugins on purpose.

The trap

`from prices import cents` does not bind `prices`. Reaching for `prices.MENU` looks natural and is a `NameError`.

Save as `missing_module_name.py`:

```
# missing_module_name.py
from prices import cents


def main():
    print(cents("latte"))
    try:
        print(prices.MENU)
    except NameError as e:
        print(type(e).__name__ + ":", e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python missing_module_name.py
```

Output:

```
400
NameError: name 'prices' is not defined
```

The fix is `import prices` (then `prices.cents` and `prices.MENU`), or `from prices import cents, MENU`. Do not add `from prices import *`. Star imports dump unknown names into your file and make collisions invisible.

The boring rule

- One idea per module. A module name is a file name without `.py`.
- Prefer `import prices` when you will use several names. Prefer `from prices import cents` when one function *is* the API.
- Use `as` for a clash or a long name, not for decoration.
- `importlib.import_module` is for a name you did not know when you wrote the file.
- Never `from module import *` in application code.
- Library modules stay quiet at import time. Work happens in functions. The `main` guard lives in the program file.

Try this

1. Add `cookie` to `MENU` at 200 cents. Run `order.py` again and include `"cookie"` in `total`.
2. In `from_order.py`, also import `MENU` and print `sorted(MENU)`.
3. In `load_prices.py`, set `name = "no_such_desk"` and catch `ModuleNotFoundError`. Print the exception type.
4. Split `total` into its own file `totals.py` that `import prices`. Have `order.py` `import totals` and print `totals.total(["tea", "tea"])`.

Packages and init

Packages and `__init__`

A **package** is a directory of modules with a way to be imported as `desk.tickets`. The boring default is a folder named `desk/` with an `__init__.py` and a few sibling `.py` files. **Relative imports** (`from .tickets import Ticket`) keep the package movable. A **namespace package** is a directory *without* `__init__.py`; skip it unless you are splitting one package across locations.

Mental model

`import desk.tickets` means: find a directory `desk` on `sys.path`, treat it as a package, load `desk/tickets.py` as the submodule `desk.tickets`.

`__init__.py` runs when the package itself is first imported (`import desk`). Put re-exports there if the public names are few. Leave it empty if callers should write `desk.tickets`.

A leading `.` in `from .tickets import Ticket` means "the package I belong to." Two dots (`..`) means the parent package. Relative imports only work when the file is loaded **as part of a package**, not when you run `uv run python desk/labels.py`.

A namespace package (PEP 420) is a directory with no `__init__.py`. Python can merge several such directories of the same name on `sys.path`. Ordinary application code does not need that.

Worked examples

Create a folder `desk` next to the program you will run. Save the package files inside `desk/`. Save `show_desk.py` beside `desk/`, not inside it. Run from that parent folder.

Case 1: A directory with `__init__.py`

Save as `desk/tickets.py`:

```
# desk/tickets.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table
```

Save as `desk/__init__.py`. Importing `desk` re-exports `Ticket`.

```
# desk/__init__.py
from .tickets import Ticket

__all__ = ["Ticket"]
```

Save as `show_desk.py`:

```
# show_desk.py
import desk


def main():
    t = desk.Ticket(7, 12)
    print(t.id, t.table)
    print(desk.__all__)


if __name__ == "__main__":
    main()
```

Run (from the parent of `desk/`):

```
uv run python show_desk.py
```

Output:

```
7 12
['Ticket']
```

`__all__` lists the public names for `from desk import *`. You still will not star-import. The list is documentation.

Case 2: Relative imports between siblings

Save as `desk/labels.py`. The `.` is required: `tickets` is a sibling, not a top-level module.

```
# desk/labels.py
from .tickets import Ticket


def label(ticket):
    return f"ticket {ticket.id} → table {ticket.table}"
```

Save as `show_label.py` next to `desk/`:

```
# show_label.py
from desk.labels import label
from desk.tickets import Ticket


def main():
    print(label(Ticket(7, 12)))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python show_label.py
```

Output:

```
ticket 7 → table 12
```

Absolute `from desk.tickets import Ticket` also works inside the package. Relative imports stay correct if you later rename the top folder's *path* but keep the package structure. Pick one style per package and stick to it. Relative is the usual choice among siblings.

Case 3: A namespace package, briefly

Save as `tools/clock.py`. There is **no** `tools/__init__.py`.

```
# tools/clock.py
def now():
    return "09:00"
```

Save as `show_tools.py` next to `tools/`:

```
# show_tools.py
import tools.clock
import tools


def main():
    print(tools.clock.now())
    print(tools.__file__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python show_tools.py
```

Output:

```
09:00
None
```

`tools` imported. `__file__` is `None` because a namespace package is a list of locations, not one source file. For application code, add `__init__.py` anyway. You want a regular package with a single directory and a place to put re-exports.

The trap

Relative imports fail when the file is run as a script. Python then has no package context.

Keep `desk/labels.py` as in Case 2. Run it directly:

```
uv run python desk/labels.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "desk/labels.py", line 2, in <module>
    from .tickets import Ticket
ImportError: attempted relative import with no known parent package
```

The line number matches the `from .tickets` line in the file you saved. The fix is not to rewrite the import as `from tickets import Ticket` (that breaks `python -m desk` later). The fix is to run a module *as* a package: `uv run python -m desk.labels` will still do nothing useful because `labels.py` has no `main`. Put a program next to the package (`show_label.py`) or add `__main__.py` (next chapter).

The boring rule

- Application layout: `desk/__init__.py`, `desk/tickets.py`, a program beside the folder or `python -m desk`.
- `__init__.py` may be empty. Re-export only a short public list.
- Sibling imports: `from .tickets import Ticket`.
- Do not run `uv run python desk/labels.py`. You lose the package.
- Do not skip `__init__.py` to look modern. Namespace packages are for split distributions, not for your desk app.
- `__all__` is a list of strings, not a substitute for a README.

Try this

1. Add `desk/money.py` with `def with_tax(cents, rate=0.1): return round(cents * (1 + rate))`. Import it from `show_desk.py` as `from desk.money import with_tax` and print `with_tax(400)`.
2. Re-export `label` from `desk/__init__.py`. Call `desk.label(desk.Ticket(1, 2))` from a tiny program.
3. In `desk/labels.py`, try `from tickets import Ticket` (no dot), run `show_label.py`, and read the `ModuleNotFoundError`. Put the dot back.
4. Add `tools/bell.py` with `def ring(): return "ding"` and print `tools.bell.ring()` from `show_tools.py`.

Entry Points and main

Entry Points and `__main__`

An **entry point** is the first function that runs when a human (or a service) starts your program. The boring default is a `main()` plus `if __name__ == "__main__"`. For a package, add `__main__.py` so `uv run python -m desk` works. For a command people type, declare `[project.scripts]` in `pyproject.toml`.

Mental model

When you run a file, Python sets `__name__` to `"__main__"`. When another file imports it, `__name__` is the module name. The **main guard** is the `if` that calls `main` only in the first case.

`uv run python -m desk` looks for a package `desk` and runs `desk/__main__.py` (or `desk.py` if there is no package). The `-m` form keeps the package context, so relative imports work.

`[project.scripts]` in `pyproject.toml` maps a command name to `module:function`. After the project is installed into the environment, `uv run desk` calls that function with no arguments. `main` should return an `int` exit code or `None`.

Worked examples

Case 1: The main guard

Save as `open_desk.py`. Importing this file later will not print.

```
# open_desk.py
def main():
    print("desk is open")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Run:

```
uv run python open_desk.py
```

Output:

```
desk is open
```

`SystemExit` turns the return code into the process status. `0` means success.

Case 2: `python -m` on a package with `__main__.py`

Create a folder `desk/` next to nothing else you care about, and save these three files inside it. Run from the **parent** of `desk/`.

Save as `desk/__init__.py` (empty is fine; a comment is enough):

```
# desk/__init__.py
```

Save as `desk/cli.py`:

```
# desk/cli.py
def main():
    print("desk is open")
    return 0
```

Save as `desk/__main__.py`. This file is the `-m` entry.

```
# desk/__main__.py
from .cli import main

if __name__ == "__main__":
    raise SystemExit(main())
```

Run:

```
uv run python -m desk
```

Output:

```
desk is open
```

`from .cli import main` works because `-m desk` loaded a package. `cli.py` stays importable by tests.

Case 3: `[project.scripts]` in `pyproject.toml`

Put the same `desk/` package **inside a project folder** `desk_app/` with a `pyproject.toml`. Layout:

- `desk_app/pyproject.toml`
- `desk_app/desk/__init__.py`

- `desk_app/desk/cli.py`
- `desk_app/desk/__main__.py` (optional here; the script entry does not need it)

Save as `desk_app/pyproject.toml`:

```
# pyproject.toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "desk"
version = "0.1.0"
requires-python = ">=3.14"
description = "A tiny desk command"

[project.scripts]
desk = "desk.cli:main"
```

Reuse `desk/cli.py` from Case 2 (`def main():` that prints and returns 0). From `desk_app/`:

```
uv run desk
```

Output:

```
desk is open
```

`uv run` builds the project into the environment and puts a `desk` command on the path. The value `"desk.cli:main"` means "import `desk.cli`, call `main`."

You can keep `__main__.py` as well. Then both `uv run desk` and `uv run python -m desk` start the same `main`.

The trap

Running a file *inside* the package by path drops the package. Relative imports die.

Save as `desk/tickets.py`:

```
# desk/tickets.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table
```

Save as `desk/cli.py`:

```python
# desk/cli.py
from .tickets import Ticket


def main():
    t = Ticket(7, 12)
    print(f"ticket {t.id} → table {t.table}")
    return 0
```

Save as `desk/__main__.py` (same as Case 2) and an empty `desk/__init__.py`. Run the CLI file by path:

```
uv run python desk/cli.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "desk/cli.py", line 2, in <module>
    from .tickets import Ticket
ImportError: attempted relative import with no known parent package
```

The fix is not to delete the dot. Keep the relative import and start the package:

```
uv run python -m desk
```

Output:

```
ticket 7 → table 12
```

The boring rule

- Every program file: `def main():` and `if __name__ == "__main__": raise SystemExit(main())`.
- Packages people run: `__main__.py` that calls `cli.main`.
- Commands people type: `[project.scripts] name = "pkg.cli:main"`.
- Start the package with `python -m desk`, never `python desk/cli.py`.
- `main` takes no arguments when it is a console script. Parse `sys.argv` inside it, or use `argparse`.

Try this

1. In `open_desk.py`, print `__name__` inside `main`. Run the file. Then from a second file `import open_desk` (no call) and confirm it prints nothing.
2. Add `desk/tickets.py` and print a ticket label from `desk/cli.py` using a relative import. Start it with `uv run python -m desk`.

3. Change `[project.scripts]` so the command is `open-desk = "desk.cli:main"`. Run `uv run open-desk`.
4. Add a main guard to `desk/cli.py` so `uv run python -m desk.cli` also prints. Keep `__main__.py` delegating to the same `main`.

Objects

Classes and Instances

Classes and Instances

A **class** is a blueprint. An **instance** is one concrete object built from it. The boring default is a small class with `__init__` to store data, and methods that use that data. If you have no state that lives, write a function.

Mental model

`class Ticket:` creates a class object. `Ticket(7, 12)` calls `__init__` and returns an instance. `self` is that instance. Every method's first parameter is `self`. You do not pass it at the call site: `t.label()` passes `t` for you.

A **method** is a function defined on a class. On the class it is a function. On the instance it is a **bound method**: `self` is already filled in. A plain function that takes a ticket is still valid. Use a method when the behaviour belongs with the data and you will call it on many instances.

Instance attributes live on `self` (`self.id`). A name assigned on the class body is shared by all instances until an instance overrides it.

Worked examples

Case 1: `class`, `__init__`, `self`

Save as `ticket.py`. `__init__` stores two numbers. It returns `None` on purpose; the instance is created by the class machinery.

```
# ticket.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table

    def label(self):
        return f"ticket {self.id} → table {self.table}"


def main():
    t = Ticket(7, 12)
    print(t.label())
    print(t.id, t.table)
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket.py
```

Output:

```
ticket 7 → table 12
7 12
```

Two instances are two objects. Changing one table does not change the other.

Case 2: Methods versus functions

Save as `ticket_fn.py`. The same label is a method and a function. Both are correct. The method travels with the class. The function works on anything with `.id` and `.table`.

```
# ticket_fn.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table

    def label(self):
        return f"ticket {self.id} → table {self.table}"


def label_ticket(ticket):
    return f"ticket {ticket.id} → table {ticket.table}"


def main():
    t = Ticket(7, 12)
    print(t.label())
    print(label_ticket(t))
    print(Ticket.label(t))
    print(type(t.label).__name__)
    print(type(Ticket.label).__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_fn.py
```

Output:

```
ticket 7 → table 12
ticket 7 → table 12
ticket 7 → table 12
method
function
```

`t.label()` is the everyday call. `Ticket.label(t)` is the same function with `self` passed by hand. Prefer `t.label()` in application code.

Case 3: Several instances

Save as `shift_tickets.py`. A shift holds a list of tickets. The desk object does not need a hierarchy; it needs a list.

```
# shift_tickets.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table

    def label(self):
        return f"ticket {self.id} → table {self.table}"


class Shift:
    def __init__(self, name):
        self.name = name
        self.tickets = []

    def add(self, ticket):
        self.tickets.append(ticket)

    def dump(self):
        print(self.name)
        for t in self.tickets:
            print(" ", t.label())


def main():
    morning = Shift("morning")
    morning.add(Ticket(1, 4))
```

```
    morning.add(Ticket(2, 12))
    morning.dump()
    print("count", len(morning.tickets))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_tickets.py
```

Output:

```
morning
  ticket 1 → table 4
  ticket 2 → table 12
count 2
```

`Shift` has state (`tickets`) and behaviour that belongs to it (`add`, `dump`). That is enough reason for a class.

The trap

A mutable assigned in the class body is **one** object, shared by every instance. Two tickets, one notes list.

Save as `shared_notes.py`:

```
# shared_notes.py
class Ticket:
    notes = []

    def __init__(self, id):
        self.id = id

    def add_note(self, text):
        self.notes.append(text)


def main():
    a = Ticket(1)
    b = Ticket(2)
    a.add_note("window")
    print("a", a.notes)
    print("b", b.notes)
```

```
    print("same", a.notes is b.notes)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shared_notes.py
```

Output:

```
a ['window']
b ['window']
same True
```

The fix is an instance list in `__init__`:

```
# own_notes.py
class Ticket:
    def __init__(self, id):
        self.id = id
        self.notes = []

    def add_note(self, text):
        self.notes.append(text)


def main():
    a = Ticket(1)
    b = Ticket(2)
    a.add_note("window")
    print("a", a.notes)
    print("b", b.notes)
    print("same", a.notes is b.notes)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python own_notes.py
```

Output:

```
a ['window']
```

```
b []
same False
```

Class attributes are fine for constants (`kind = "ticket"`). They are not fine for lists you append to.

The boring rule

- Class = data that lives + behaviour that belongs to it. Otherwise write a function.
- `__init__` stores attributes on `self`. It does not return the instance.
- Call methods on the instance: `t.label()`.
- Put mutable per-instance state in `__init__`, not on the class body.
- Two instances are two objects. Do not reuse one ticket as a "template" by mutating it in place unless that is the real lifecycle.

Try this

1. In `ticket.py`, add a method `move(self, table)` that sets `self.table`. Print `label` before and after a move.
2. In `ticket_fn.py`, add a function `move_ticket(ticket, table)` and a method `move`. Show both.
3. In `shift_tickets.py`, add `Shift.tables` that returns a list of table numbers in ticket order.
4. In `shared_notes.py`, move `notes = []` into `__init__` as `self.notes = []` and confirm `b.notes` stays empty.

Inheritance and Composition

Inheritance and Composition

Composition means an object *has* another object and asks it to do work. **Inheritance** means an object *is* a more specific kind of another object. The boring default is composition. Inherit when the English sentence "X is a Y" stays true as the code grows, and call `super()` to run the parent's setup.

Mental model

`Desk` has a `Register` and a `Clock`. That is composition: attributes that are objects. When the register's rules change, the desk still looks like a desk.

`PaidTicket` is a `Ticket` with an amount. That is inheritance: the child can sit anywhere a ticket sits, plus it knows `cents`. `class PaidTicket(Ticket):` puts `Ticket` in the child's method search. `super().__init__(...)` runs the parent `__init__` so you do not copy `self.id = id`.

If you inherit only to reuse a couple of methods, you are borrowing with a bad contract. Put the shared helper on a third object, or make it a function.

Worked examples

Case 1: Composition (the default)

Save as `desk_parts.py`. The desk does not *is-a* register. It *has* one.

```
# desk_parts.py
class Register:
    def __init__(self):
        self.cents = 0

    def take(self, n):
        self.cents += n
        return self.cents


class Clock:
    def __init__(self, hour):
        self.hour = hour

    def stamp(self):
```

```
        return f"{self.hour:02d}:00"


class Desk:
    def __init__(self, register, clock):
        self.register = register
        self.clock = clock

    def ring_up(self, cents):
        total = self.register.take(cents)
        return f"{self.clock.stamp()} took {cents} now {total}"


def main():
    desk = Desk(Register(), Clock(9))
    print(desk.ring_up(400))
    print(desk.ring_up(250))
    print("in register", desk.register.cents)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_parts.py
```

Output:

```
09:00 took 400 now 400
09:00 took 250 now 650
in register 650
```

You can pass a fake register in a test. That is the payoff. Inheritance would glue the desk to one register class.

Case 2: Inheritance when it is a true is-a

Save as `paid_ticket.py`. A paid ticket is a ticket. It still has `id` and `table`. It also has `cents`. `label` extends the parent's label.

```
# paid_ticket.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table
```

```
    def label(self):
        return f"ticket {self.id} → table {self.table}"


class PaidTicket(Ticket):
    def __init__(self, id, table, cents):
        super().__init__(id, table)
        self.cents = cents

    def label(self):
        return f"{super().label()} paid {self.cents}c"


def main():
    unpaid = Ticket(7, 12)
    paid = PaidTicket(8, 4, 650)
    print(unpaid.label())
    print(paid.label())
    print(isinstance(paid, Ticket))
    print(paid.table)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python paid_ticket.py
```

Output:

```
ticket 7 → table 12
ticket 8 → table 4 paid 650c
True
4
```

`isinstance(paid, Ticket)` is true. Code that accepts a ticket can accept a paid ticket if it only needs `label`, `id`, and `table`.

Case 3: `super()` in a short chain

Save as `shift_ticket.py`. `ShiftTicket` is still a ticket. It adds a shift name. `super()` is the parent, not a copy-paste of `__init__`.

```
# shift_ticket.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table

    def label(self):
        return f"ticket {self.id} → table {self.table}"


class ShiftTicket(Ticket):
    def __init__(self, id, table, shift):
        super().__init__(id, table)
        self.shift = shift

    def label(self):
        return f"{self.shift} {super().label()}"


def main():
    t = ShiftTicket(3, 11, "morning")
    print(t.label())
    print(t.id, t.table, t.shift)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_ticket.py
```

Output:

```
morning ticket 3 → table 11
3 11 morning
```

If `Ticket.__init__` later grows a `notes` list, `ShiftTicket` still gets it. That is why you call `super()` instead of setting `self.id` again.

The trap

Inheriting for reuse when the relationship is *has-a*. `Desk(Register)` makes every desk a register. Callers can `desk.take(400)` and skip the clock. Tests cannot swap the register.

Save as `desk_is_register.py`:

```
# desk_is_register.py
class Register:
    def __init__(self):
        self.cents = 0

    def take(self, n):
        self.cents += n
        return self.cents


class Desk(Register):
    def __init__(self, hour):
        super().__init__()
        self.hour = hour

    def ring_up(self, cents):
        total = self.take(cents)
        return f"{self.hour:02d}:00 took {cents} now {total}"


def main():
    desk = Desk(9)
    print(desk.ring_up(400))
    print("also a register:", isinstance(desk, Register))
    desk.take(50)
    print("sneaky total", desk.cents)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_is_register.py
```

Output:

```
09:00 took 400 now 400
also a register: True
sneaky total 450
```

It runs. It is the wrong shape. Use `desk_parts.py`: a desk **has** a register. `take` stays on the register. The desk's public method is `ring_up`.

The boring rule

- Default to composition: store the helper on `self` and call it.
- Inherit only for a real is-a that you are willing to keep.
- Call `super().__init__(...)` in the child. Do not copy parent attributes by hand.
- Override a method to extend behaviour; call `super().method(...)` when the parent's work still applies.
- `isinstance` on your own classes is a smell if you need it in many places. Prefer a method on the object.
- Do not build a diamond of mixins to save ten lines.

Try this

1. In `desk_parts.py`, add `Printer` with `def emit(self, text): return text`. Compose it on `Desk` and print `ring_up` through the printer.
2. In `paid_ticket.py`, add `VoidedTicket(Ticket)` with `label` returning the parent label plus `voided`. Keep `super().__init__`.
3. In `shift_ticket.py`, drop `super().__init__` and set `self.id` by hand. Then add `self.notes = []` to `Ticket.__init__` and notice `ShiftTicket` does not get it. Put `super()` back.
4. Rewrite `desk_is_register.py` as composition (copy the shape of Case 1). Confirm `isinstance(desk, Register)` is `False`.

Properties and Descriptors

Properties and Descriptors

A **property** is an attribute that runs a method when you read it (and optionally when you set it). The boring default is `@property` for a computed value or a checked assignment. A **descriptor** is the general protocol behind properties. Write a custom descriptor only when the same rule must live on several classes.

Mental model

`t.table` looks like data. If `table` is a property, Python calls a method instead of returning a dict entry. Callers still write `t.table` and `t.table = 4`. They do not write `t.get_table()`.

`@property` turns a method into the getter. `@table.setter` registers the setter. Store the real value on `self._table` (leading underscore: "internal"). The getter returns it. The setter validates, then assigns `_table`. If the setter assigns `self.table`, it calls itself until the stack blows.

A descriptor is a class with `__get__` and usually `__set__`. Putting an instance of it on a class body makes that name a managed attribute. `property` is a descriptor. You almost never need a second one.

Worked examples

Case 1: `@property` for a computed total

Save as `order_total.py`. `total` is not stored. It is derived from `cents` and `tax_rate`. There is no setter: callers cannot assign `order.total = 0` by accident.

```
# order_total.py
class Order:
    def __init__(self, cents, tax_rate=0.1):
        self.cents = cents
        self.tax_rate = tax_rate

    @property
    def total(self):
        return round(self.cents * (1 + self.tax_rate))


def main():
    order = Order(400)
```

```
    print(order.total)
    order.cents = 500
    print(order.total)
    try:
        order.total = 0
    except AttributeError as e:
        print(type(e).__name__ + ":", e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python order_total.py
```

Output:

```
440
550
AttributeError: property 'total' of 'Order' object has no setter
```

Computed fields stay properties. Do not cache them until you have measured a problem.

Case 2: A setter that validates

Save as `ticket_table.py`. Assigning `table` in `__init__` goes through the setter, so `Ticket(1, 0)` fails the same way as a later assignment.

```
# ticket_table.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table

    @property
    def table(self):
        return self._table

    @table.setter
    def table(self, n):
        if n <= 0:
            raise ValueError(f"table {n}: must be positive")
        self._table = n
```

```
def main():
    t = Ticket(7, 12)
    print(t.table)
    t.table = 4
    print(t.table)
    try:
        t.table = 0
    except ValueError as e:
        print(type(e).__name__ + ":", e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_table.py
```

Output:

```
12
4
ValueError: table 0: must be positive
```

One rule, two paths (`__init__` and later). That is why the setter is worth it.

Case 3: A tiny descriptor

Save as `positive.py`. `Positive` is the same "must be > 0" rule, reusable on more than one name. `__set_name__` records the attribute name so storage can be `_table` or `_cents`.

```
# positive.py
class Positive:
    def __set_name__(self, owner, name):
        self.private = "_" + name

    def __get__(self, obj, owner):
        if obj is None:
            return self
        return getattr(obj, self.private)

    def __set__(self, obj, value):
        if value <= 0:
            raise ValueError(f"{self.private[1:]} must be positive")
        setattr(obj, self.private, value)
```

```
class Ticket:
    table = Positive()
    cents = Positive()

    def __init__(self, id, table, cents):
        self.id = id
        self.table = table
        self.cents = cents


def main():
    t = Ticket(7, 12, 400)
    print(t.table, t.cents)
    try:
        t.cents = 0
    except ValueError as e:
        print(type(e).__name__ + ":", e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python positive.py
```

Output:

```
12 400
ValueError: cents must be positive
```

Two fields, one rule. That is the moment a descriptor earns a file. For a single field, Case 2 is enough.

The trap

A setter that assigns the public name calls itself. RecursionError. The same bug happens if you write `self.table = n` inside `table.setter` instead of `self._table = n`.

Save as `setter_loop.py`:

```
# setter_loop.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self._table = table
```

```
    @property
    def table(self):
        return self._table

    @table.setter
    def table(self, n):
        if n <= 0:
            raise ValueError(f"table {n}: must be positive")
        self.table = n


def main():
    t = Ticket(7, 12)
    try:
        t.table = 4
    except RecursionError as e:
        print(type(e).__name__ + ":", e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python setter_loop.py
```

Output:

```
RecursionError: maximum recursion depth exceeded
```

`__init__` assigns `_table` directly, so construction survives. The later `t.table = 4` dies. Fix: `self._table = n` in the setter, and in `__init__` prefer `self.table = table` so construction uses the same check (see `ticket_table.py`).

A second trap: writing `Positive` for one field on one class. Use `@property`. Save the descriptor for a repeated rule.

The boring rule

- `@property` for derived values. No setter unless callers must assign.
- `@x.setter` for validation. Store on `self._x`.
- Construction should go through the setter (`self.x = x` in `__init__`) so bad values fail early.
- A custom descriptor is for one rule on many attributes or many classes.
- Prefer a property over a custom descriptor. Prefer a function over a property if the name is clearly an action (`label()`, not `.label` as data).
- Do not put business logic in `__get__` that hits the network. Keep descriptors dull.

Try this

1. In `order_total.py`, add `@property def tax(self)` that returns `self.total - self.cents`. Print it for 400 cents.
2. In `ticket_table.py`, construct `Ticket(1, 0)` and catch `ValueError`.
3. In `positive.py`, add `id = Positive()` and try `Ticket(0, 12, 400)`. Read the error.
4. Fix `setter_loop.py` so `t.table = 4` prints `4`. Keep the `<= 0` check.

Dunder Methods and the Data Model

Dunder Methods and the Data Model

Python's operators and builtins are method calls with nicer spelling. `len(x)` is `x.__len__()`. `a == b` is `a.__eq__(b)`. `print(x)` uses `x.__repr__()` unless you also wrote `__str__`. The boring default is to implement the few methods callers actually use — not a complete "numeric type" or a fake sequence.

Mental model

A **dunder** (double underscore) method is a hook the language looks up by name. The set of those hooks is the **data model**: how your type talks to `for`, `+`, `repr`, containers, and so on.

You do not register anything. You define the method. Python calls it.

Return `NotImplemented` (the singleton, not an exception) when an operator does not know the other type. Python can then try the other operand. Returning `False` from `__eq__` for every foreign object shuts that down.

Do not implement a dunder because a list of magic methods looks thorough. Each one is an API you now have to keep honest.

Worked examples

Case 1: `__repr__` that looks like a constructor

Save as `ticket_repr.py`. One debug form is enough for a desk record. Skip `__str__` until you have a sentence for a person.

```
# ticket_repr.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table

    def __repr__(self):
        return f"Ticket(id={self.id!r}, table={self.table!r})"


def main():
    t = Ticket(7, 12)
```

```
    print(t)
    print(repr(t))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_repr.py
```

Output:

```
Ticket(id=7, table=12)
Ticket(id=7, table=12)
```

`print` uses `__str__` if you wrote it, otherwise `__repr__`. `{self.id!r}` puts quotes around strings so a table named `12` and a table numbered `12` stay distinguishable.

Case 2: `__eq__` for the identity you mean

Save as `ticket_eq.py`. Two tickets with the same id are the same ticket even if the table changed.

```
# ticket_eq.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table

    def __repr__(self):
        return f"Ticket(id={self.id!r}, table={self.table!r})"

    def __eq__(self, other):
        if not isinstance(other, Ticket):
            return NotImplemented
        return self.id == other.id


def main():
    a = Ticket(7, 12)
    b = Ticket(7, 4)
    c = Ticket(8, 12)
    print(a == b)
    print(a == c)
    print(a == 7)
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_eq.py
```

Output:

```
True
False
False
```

`a == 7` is `False` because `Ticket` returns `NotImplemented` and `int` does not know tickets. That is the right shape. Do not special-case `int` here.

Case 3: `__len__` and `__iter__` for a small collection

Save as `shift_board.py`. If the object *is* a list of shifts, let `len` and `for` work. You do not need `__getitem__` until someone indexes it.

```
# shift_board.py
class Shift:
    def __init__(self, name, hours):
        self.name = name
        self.hours = hours

    def __repr__(self):
        return f"Shift({self.name!r}, {self.hours})"


class ShiftBoard:
    def __init__(self, shifts):
        self._shifts = list(shifts)

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

    def __iter__(self):
        return iter(self._shifts)


def main():
    board = ShiftBoard(
        [
            Shift("open", 4),
```

```
            Shift("mid", 6),
            Shift("close", 5),
        ]
    )
    print(len(board))
    for shift in board:
        print(shift)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_board.py
```

Output:

```
3
Shift('open', 4)
Shift('mid', 6)
Shift('close', 5)
```

`__iter__` can return `iter(self._shifts)`. Do not hand-write a custom iterator class for a list you already own.

Case 4: one operator, for an amount

Save as `order_cents.py`. Money-as-cents is a type that can add to itself. It cannot add to a bare `int` until you decide what that means.

```
# order_cents.py
class Cents:
    def __init__(self, amount):
        if amount < 0:
            raise ValueError("cents cannot be negative")
        self.amount = amount

    def __repr__(self):
        return f"Cents({self.amount})"

    def __add__(self, other):
        if not isinstance(other, Cents):
            return NotImplemented
        return Cents(self.amount + other.amount)
```

```
    def __eq__(self, other):
        if not isinstance(other, Cents):
            return NotImplemented
        return self.amount == other.amount


def main():
    soup = Cents(450)
    bread = Cents(200)
    total = soup + bread
    print(total)
    print(total == Cents(650))
    try:
        soup + 200
    except TypeError as e:
        print(type(e).__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python order_cents.py
```

Output:

```
Cents(650)
True
TypeError
```

Skip `__iadd__`, `__radd__`, `__sub__`, `__mul__` until a caller needs them. `__add__` already covers `total = soup + bread`. In-place `+=` will use `__add__` and rebind the name.

The trap

Two common overreaches: silent attributes, and equality that breaks sets.

Save as `noisy_ticket.py`. `__getattr__` runs for **missing** names. A typo becomes a string instead of `AttributeError`.

```
# noisy_ticket.py
class Ticket:
    def __init__(self, id, table):
        self.id = id
        self.table = table
```

```
    def __bool__(self):
        return self.table != 0

    def __getattr__(self, name):
        return f"missing:{name}"


def main():
    t = Ticket(7, 12)
    print(bool(t))
    print(t.table)
    print(t.tabl)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python noisy_ticket.py
```

Output:

```
True
12
missing:tabl
```

`t.table` still works (`__getattr__` is not used when the attribute exists). `t.tabl` does not fail. Delete `__getattr__`. Let the typo raise.

The other overreach: `__eq__` without a story for hashing.

```
# unhashable.py
class Ticket:
    def __init__(self, id):
        self.id = id

    def __eq__(self, other):
        if not isinstance(other, Ticket):
            return NotImplemented
        return self.id == other.id


def main():
    try:
        {Ticket(7), Ticket(8)}
    except TypeError as e:
```

```
        print(type(e).__name__)
        print(e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python unhashable.py
```

Output:

```
TypeError
cannot use 'Ticket' as a set element (unhashable type: 'Ticket')
```

User-defined `__eq__` makes the type unhashable unless you also define `__hash__`. That is a feature: mutable records should not be set elements. Key a dict by `ticket.id` instead.

The boring rule

- Write `__repr__` first. Make it look like a call that would rebuild the object.
- Write `__eq__` only when value equality is real. Return `NotImplemented` for other types.
- Write `__len__` and `__iter__` when the object *is* a collection. Stop there until indexing is a real need.
- Write `__add__` (and friends) only for amounts. One operator is a type. Twelve operators are a science project.
- Do not implement `__getattr__`, `__bool__`, or `__str__` "while you are here."
- If you write `__eq__`, either keep the type out of sets or define `__hash__` on an immutable value. Default: dict keyed by id.

Try this

1. In `ticket_repr.py`, add `__str__` that returns `ticket 7 @ table 12`. Print both `t` and `repr(t)`.
2. In `shift_board.py`, add a function `total_hours(board)` that loops with `for` and sums `shift.hours`.
3. In `order_cents.py`, add `__mul__` so `Cents(450) * 2` is `Cents(900)`. Reject a non-`int` with `NotImplemented`.
4. Remove `__getattr__` from `noisy_ticket.py` and confirm `t.tabl` raises `AttributeError`.

Enums, slots, and Dataclasses

Enums, slots, and Dataclasses

Three tools for data that has outgrown a dict: an **Enum** for a closed set of names, a **dataclass** for a record, and **slots** as a layout opt-in. The boring default is a dataclass **without** slots. Add an Enum when a string status starts getting typos. Add slots when a profiler says the extra `__dict__` on millions of rows is the bill.

Mental model

`enum.Enum` members are singletons. `Status.OPEN is Status.OPEN` is true. Construction from a value (`Status("open")`) either returns that member or raises `ValueError`. That is the point: `"payed"` cannot sneak through.

`@dataclass` writes `__init__`, `__repr__`, and `__eq__` from annotated fields. The annotations here are field declarations, not a typing chapter. You already met dataclasses as records; this chapter is the knobs: `frozen=True`, `slots=True`, and when to leave both off.

Slots replace the per-instance `__dict__` with a fixed set of attributes. You cannot assign `t.note = "window"` later. That saves memory and catches typos. It also gets in the way of mixins, optional fields, and debugging. Measure first.

Worked examples

Case 1: a closed kitchen status

Save as `ticket_status.py`. Compare members with `is`. Read the stored string with `.value` only at the edge (print, JSON).

```
# ticket_status.py
from enum import Enum


class Status(Enum):
    OPEN = "open"
    FIRED = "fired"
    PAID = "paid"


def label(status):
    return f"kitchen: {status.value}"
```

```
def main():
    s = Status.OPEN
    print(s)
    print(s is Status.OPEN)
    print(label(s))
    try:
        Status("lost")
    except ValueError as e:
        print(e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_status.py
```

Output:

```
Status.OPEN
True
kitchen: open
'lost' is not a valid Status
```

Do not loop over string literals in business code. Loop over `Status` if you need every member.

Case 2: a dataclass record

Save as `order_row.py`. Field types tell `@dataclass` what `__init__` accepts. Equality is by value.

```
# order_row.py
from dataclasses import dataclass


@dataclass
class Order:
    id: int
    table: int
    cents: int


def main():
    a = Order(1, 12, 850)
    b = Order(1, 12, 850)
```

```
    print(a)
    print(a == b)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python order_row.py
```

Output:

```
Order(id=1, table=12, cents=850)
True
```

This is the default you want for desk rows: mutable, repr for free, no slots.

Case 3: freeze when mutation is a bug

Save as `frozen_shift.py`. A published shift card should not grow an extra hour because a later line assigned `s.hours = 5`.

```
# frozen_shift.py
from dataclasses import dataclass


@dataclass(frozen=True)
class Shift:
    name: str
    hours: int


def main():
    s = Shift("open", 4)
    print(s)
    try:
        s.hours = 5
    except AttributeError as e:
        print(type(e).__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python frozen_shift.py
```

Output:

```
Shift(name='open', hours=4)
FrozenInstanceError
```

`FrozenInstanceError` is an `AttributeError`. Catch `AttributeError` if you must; better: do not assign. Frozen values can be dict keys if the fields themselves are hashable.

Case 4: slots when you opt in

Save as `slotted_ticket.py`. `slots=True` drops `__dict__`. Extra attributes fail.

```
# slotted_ticket.py
from dataclasses import dataclass


@dataclass(slots=True)
class Ticket:
    id: int
    table: int


def main():
    t = Ticket(7, 12)
    print(t)
    print(hasattr(t, "__dict__"))
    try:
        t.note = "window"
    except AttributeError as e:
        print(type(e).__name__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python slotted_ticket.py
```

Output:

```
Ticket(id=7, table=12)
False
AttributeError
```

The same dataclass **without** slots still reprs, and it lets you hang a note on the instance — convenient, and a source of "where did this attribute come from?"

```
# extra_field.py
from dataclasses import dataclass


@dataclass
class Ticket:
    id: int
    table: int


def main():
    t = Ticket(7, 12)
    t.note = "window"
    print(t)
    print(t.note)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python extra_field.py
```

Output:

```
Ticket(id=7, table=12)
window
```

`print(t)` does not show `note`. The extra field lives only on `__dict__`. If you needed `note`, it should have been a field.

The trap

String statuses fail closed in the wrong direction: they fail **open**. A typo is just another string.

```
# status_typo.py
def is_paid(status):
    return status == "paid"


def main():
    print(is_paid("payed"))
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python status_typo.py
```

Output:

```
False
```

The kitchen never marks the ticket paid. No exception. Use `Status.PAID`.

The other trap is putting `slots=True` on every dataclass because a post said it is faster. Slots and multiple inheritance fight. Slots and "I'll add a cache attribute later" fight. Frozen plus slots plus a cached property is how you end up calling `object.__setattr__`. Leave slots off until a measurement names this class.

The boring rule

- `Enum` for a handful of named states. Compare members, not `.value`, inside the program.
- `@dataclass` for records. Field annotations are the constructor.
- `@dataclass(frozen=True)` when the value is a key or accidental assignment would be a defect.
- No `slots=True` until you have many instances and a profile. The default dataclass is the default.
- Do not invent extra attributes on instances. Add a field.
- Do not mix a hand-written `__init__` with `@dataclass` unless you have read what `field()` and `__post_init__` are for — and you still might not need them.

Try this

1. Add `Status.VOID = "void"` and a `label` branch that prints `voided` for that member.
2. Make `Order` frozen. Try to add `a.cents = 0` and print the exception type.
3. Combine `@dataclass(frozen=True, slots=True)` on `Shift`. Confirm you still cannot set `hours`, and that there is no `__dict__`.
4. Replace the string in `status_typo.py` with `Status`. Pass `Status.PAID` and a wrong construction (`Status("payed")`) and show the `ValueError`.

Typing

Why Types Matter

Why Types Matter

Python still runs without annotations. A type checker does not. **Gradual typing** means you add hints at the seams that hurt — public functions, `None` returns, dicts of ids — and leave the rest alone until a bug names them. The boring default is: annotate the function, run the program, let **mypy**, **ty**, or **pyright** complain in CI. Do not expect the runtime to enforce those comments.

This is the first chapter in the book that uses annotations on purpose.

Mental model

An annotation is a hint attached to a parameter, a return, or a variable:

```
def label(table: int, seats: int) -> str:
    ...
```

CPython stores that hint. It does not type-check the call. `label(12, "4")` still runs. The string `"4"` is not an `int`; an f-string will still interpolate it.

A **checker** reads the hints and the call sites. **mypy**, **ty** (Astral's checker), and **pyright** (the engine behind Pylance) all do this job. They catch:

- passing a `str` where you declared `int`
- using a value that might be `None` without a check
- returning the wrong type
- treating a `dict[str, int]` as if the keys were `int`

They do not catch logic. `seats + 1` on the wrong table still looks fine if both are `int`.

Gradual means mixed files are allowed. An unannotated function is `Any` from the checker's point of view. That is why a fully untyped script is silent — and why a typed boundary next to an untyped helper leaks `Any`.

Worked examples

Case 1: a complete typed function

Save as `table_label.py`. Parameters and the return are annotated. `main` returns `None` because it only prints.

```
# table_label.py
def label(table: int, seats: int) -> str:
    return f"table {table} seats {seats}"


def main() -> None:
    print(label(12, 4))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python table_label.py
```

Output:

```
table 12 seats 4
```

The program is still ordinary Python. The hints document the contract a teammate (and a checker) can read.

Case 2: the bug a checker sees and the runtime does not

Save as `seats_bug.py`. `raw` is a string. `label` asked for `int`. Python concatenates it into the message anyway.

```
# seats_bug.py
def label(table: int, seats: int) -> str:
    return f"table {table} seats {seats}"


def main() -> None:
    raw = "4"
    print(label(12, raw))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python seats_bug.py
```

Output:

```
table 12 seats 4
```

A checker reports that argument 2 is `str`, expected `int`. The printed line looks fine, so this is easy to ship. `uv run python` is not a type checker. Run mypy, ty, or pyright on the file when you care.

Fix: `seats = int(raw)` before the call, or give `label` a `str` if that is really the input.

Case 3: `None` is a type, not a vibe

Save as `ticket_lookup.py`. `dict.get` returns the value or `None`. The return type is `int | None`. `announce` wants an `int`. Check before you call.

```python
# ticket_lookup.py
def find_table(tickets: dict[int, int], ticket_id: int) -> int | None:
    return tickets.get(ticket_id)


def announce(table: int) -> str:
    return f"now seating table {table}"


def main() -> None:
    tickets = {7: 12, 8: 4}
    table = find_table(tickets, 9)
    if table is None:
        print("ticket not on the board")
        return
    print(announce(table))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_lookup.py
```

Output:

```
ticket not on the board
```

If you skip the `None` check and pass `table` into `announce`, a checker flags it. At runtime you would print `now seating table None` — still a successful process, still wrong.

After `if table is None: return`, checkers **narrow** the type: `table` is `int` on the next line.

Case 4: annotations are not a runtime guard

Save as `no_enforcement.py`. Two strings go into a function that promised `int`. `+` concatenates.

```
# no_enforcement.py
def add_cents(a: int, b: int) -> int:
    return a + b


def main() -> None:
    print(add_cents("4", "50"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python no_enforcement.py
```

Output:

```
450
```

That is `"4" + "50"`, not 54 cents. Hints did not save the till. A checker would refuse the call. A test would refuse the result. The runtime smiled.

The trap

Annotating every local, every loop variable, and a 12-line script because a style guide said "100% coverage." You spend the afternoon arguing with the checker about a `print`. Gradual typing is a budget: spend it on functions other people call and on values that can be `None`.

The other trap is treating a checker as optional forever. Unannotated code is not "simple." It is unchecked. When the desk board grows a second module, add a checker to the same place you already run `ruff` and `pytest`.

Do not add a runtime type-validation library on day one to "make hints real." Parse at the edge (`int(raw)`), keep the core as Python, and let the checker watch the core.

The boring rule

- Annotate public functions: parameters and return. `-> None` for `main`.
- Use `X | None` whenever `None` is a real result. Check it before use.
- Run mypy, ty, or pyright in CI once the project is more than one file. Pick one checker and keep it.

- Do not expect CPython to raise because a hint was wrong.
- Do not annotate everything. Annotate the contract.
- Built-in generics (`list[int]`, `dict[int, int]`) are enough to start. The next chapter fills in the rest.

Try this

1. In `seats_bug.py`, convert `raw` with `int` before calling `label`. Confirm the program still prints the same line.
2. In `ticket_lookup.py`, look up ticket `7` instead of `9`. Print `announce(table)` after the `None` check.
3. Change `add_cents` to return `a + b` only after `isinstance` checks — then delete those checks. Prefer a checker plus `int()` at the edge over defensive `isinstance` in the core.
4. Point whichever checker you installed at `seats_bug.py` and read the error. Do not change the checker's config yet; read the default.

Type Hints in Practice

Type Hints in Practice

Python 3.14 writes collection types with the builtins: `list[int]`, `dict[str, int]`, `int | None`. You do not import `List`, `Dict`, or `Optional` for new code. A **type alias** names a meaning (`Cents`, `TableId`) so the hint reads like the desk. The boring default is those builtins plus `| None`. Leave `Any` for the rare untyped boundary.

Mental model

A hint is a type expression:

You mean	You write
list of ints	`list[int]`
dict, string keys, int values	`dict[str, int]`
int or missing	`int | None`
a named int	`type Cents = int` or `Cents: TypeAlias = int`

`X | Y` is a **union**. `None` in a union is optional in the English sense: the value may be absent. The older spelling `Optional[int]` is `int | None`. Prefer the pipe.

Aliases do not create new runtime types. `Cents` is still `int`. They exist so `price: Cents` is not a comment you forget to update.

This chapter stays on builtins plus `TypeAlias`. No `TypedDict`, no `Literal`, no `Annotated` until a later need names them.

Worked examples

Case 1: `list[int]`

Save as `shift_hours.py`. The function only accepts a list of hours. A list of names is a checker error even if you never append here.

```
# shift_hours.py
def total_hours(hours: list[int]) -> int:
    n = 0
    for h in hours:
        n += h
    return n
```

```
def main() -> None:
    print(total_hours([4, 6, 5]))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_hours.py
```

Output:

```
15
```

`tuple[int, ...]` is a tuple of ints of unknown length. `tuple[int, int]` is a pair. Use a list when the length varies.

Case 2: `dict[str, int]`

Save as `seat_counts.py`. Keys are table labels as strings. Values are seat counts.

```
# seat_counts.py
def seats_for(table: str, counts: dict[str, int]) -> int:
    return counts[table]


def main() -> None:
    counts = {"12": 4, "4": 2}
    print(seats_for("12", counts))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python seat_counts.py
```

Output:

```
4
```

`dict[int, int]` would be table **numbers**. Mixing `seats_for(12, counts)` with string keys is the bug a checker is for. Pick one key type and keep it.

Case 3: `T | None`

Save as `maybe_table.py`. `.get` returns `None` on a miss. Spell that in the return type.

```
# maybe_table.py
def table_for(ticket_id: int, board: dict[int, int]) -> int | None:
    return board.get(ticket_id)


def main() -> None:
    board = {7: 12, 8: 4}
    found = table_for(7, board)
    missing = table_for(9, board)
    print(found)
    print(missing)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python maybe_table.py
```

Output:

```
12
None
```

Callers must handle `None`. If the miss is a defect, raise `KeyError` (use `board[ticket_id]`) and return `int` instead. Do not return `None` and also raise. Pick one.

Case 4: `TypeAlias` and the `type` statement

Save as `cents_alias.py`. `TypeAlias` is the explicit typing-module spelling. It is still valid in 3.14.

```
# cents_alias.py
from typing import TypeAlias

Cents: TypeAlias = int
TableId: TypeAlias = int


def label(table: TableId, price: Cents) -> str:
    return f"table {table}: {price} cents"
```

```
def main() -> None:
    print(label(12, 850))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python cents_alias.py
```

Output:

```
table 12: 850 cents
```

The same idea with the 3.12+ statement. Prefer this in new 3.14 code.

```
# cents_type_stmt.py
type Cents = int
type TableId = int


def label(table: TableId, price: Cents) -> str:
    return f"table {table}: {price} cents"


def main() -> None:
    print(label(12, 850))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python cents_type_stmt.py
```

Output:

```
table 12: 850 cents
```

Two aliases of `int` are still interchangeable at runtime — and most checkers treat them as `int` unless you use a `NewType`. That is fine for desk code. The alias is for readers.

The trap

Mixing spellings and emptying the type.

```
# optional_mix.py
from typing import Optional


def table_for(ticket_id: int, board: dict[int, int]) -> Optional[int]:
    return board.get(ticket_id)


def main() -> None:
    print(table_for(7, {7: 12}))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python optional_mix.py
```

Output:

```
12
```

It runs. It is also the old union. In the same file, `int | None` and `Optional[int]` make the reader hunt for a difference that is not there. Pick `X | None`.

`Any` is the other leak. `def load(raw: Any) -> Any` turns the checker off. If JSON came in, parse it into `dict[str, int]` (or a dataclass) at the edge. Do not pass `Any` through the kitchen.

`list` without a parameter means "list of anything" to some checkers and "please parameterize" to others. Write `list[int]`.

The boring rule

- `list[T]`, `dict[K, V]`, `set[T]`, `tuple[T, ...]` — builtins, not `typing.List`.
- Absent values: `T | None`. Not `Optional[T]` in new code.
- Name repeated ints with `type Cents = int` (or `TypeAlias` if you are matching older files).
- Do not import `List`, `Dict`, `Optional` for new hints.
- Do not use `Any` except at a true untyped boundary, and keep that boundary thin.
- One key type per dict. String labels and integer ids are different dicts.

Try this

1. Change `shift_hours.py` so it also prints `total_hours([])`. Keep the return type `int`.
2. Add a function `tables(counts: dict[str, int]) -> list[str]` that returns `list(counts)` and print it.
3. Rewrite `optional_mix.py` to use `int | None` and delete the `typing` import.
4. Add `type TicketId = int` to `maybe_table.py` and use it on `ticket_id` and the dict keys.

Protocols and ABCs

Protocols and ABCs

A **protocol** is a shape: "this object has `send`." An **ABC** (abstract base class) is a family: "this object **declares** it is a `TicketSink`." The boring default for a desk seam is a small `Protocol`. Use an ABC when you want a shared base, `isinstance` that requires inheritance, or a place to put real helper methods. Do not build a class tree so a test can pass a fake.

Mental model

Python was always duck typed: if it has `send`, you can call `send`. **Structural typing** is that idea written down. `typing.Protocol` lists methods. A checker accepts any object that has them, whether or not it subclasses the protocol.

Nominal typing is inheritance. `abc.ABC` with `@abstractmethod` refuses to construct a subclass that forgot the method. `isinstance(x, TicketSink)` is true only if `TicketSink` is in the type's MRO (or was registered). A printer that happens to have `send` is not an ABC sink until it subclasses.

	Protocol	ABC
Match	structure (methods present)	name (subclass)
Tests	a fake with the methods is enough	fake must inherit or register
`isinstance`	only if `@runtime_checkable`	yes
Forgotten method	checker error; runtime still constructs	`TypeError` on `Broken()`

Keep the protocol tiny. One method is a seam. Twelve methods is a god object with extra steps.

Worked examples

Case 1: a `TicketSink` protocol

Save as `ticket_sink_protocol.py`. `Printer` and `MemorySink` do not inherit. Both satisfy `TicketSink` because they have `send`.

```
# ticket_sink_protocol.py
from typing import Protocol
```

```
class TicketSink(Protocol):
    def send(self, ticket_id: int, message: str) -> None: ...


class Printer:
    def send(self, ticket_id: int, message: str) -> None:
        print(f"#{ticket_id} {message}")


class MemorySink:
    def __init__(self) -> None:
        self.lines: list[str] = []

    def send(self, ticket_id: int, message: str) -> None:
        self.lines.append(f"#{ticket_id} {message}")


def fire(sink: TicketSink, ticket_id: int) -> None:
    sink.send(ticket_id, "fired")


def main() -> None:
    printer = Printer()
    memory = MemorySink()
    fire(printer, 7)
    fire(memory, 8)
    print(memory.lines)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_sink_protocol.py
```

Output:

```
#7 fired
['#8 fired']
```

The `...` body is the protocol's way of saying "no implementation." A checker uses the signature. The runtime never calls `TicketSink.send`.

`MemorySink` is the fake you want in a test: it records lines. No mock library. No subclass.

Case 2: the same seam as an ABC

Save as `ticket_sink_abc.py`. `Printer` must inherit. `NotASink` has the method and still fails `isinstance`.

```
# ticket_sink_abc.py
from abc import ABC, abstractmethod


class TicketSink(ABC):
    @abstractmethod
    def send(self, ticket_id: int, message: str) -> None:
        raise NotImplementedError


class Printer(TicketSink):
    def send(self, ticket_id: int, message: str) -> None:
        print(f"#{ticket_id} {message}")


class NotASink:
    def send(self, ticket_id: int, message: str) -> None:
        print(f"#{ticket_id} {message}")


def fire(sink: TicketSink, ticket_id: int) -> None:
    sink.send(ticket_id, "fired")


def main() -> None:
    fire(Printer(), 7)
    print(isinstance(Printer(), TicketSink))
    print(isinstance(NotASink(), TicketSink))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_sink_abc.py
```

Output:

```
#7 fired
True
False
```

`fire(NotASink(), 7)` still **runs**. Annotations are not runtime checks. `isinstance` is the ABC's actual gate. If you needed that gate, you wanted an ABC. If you only needed a checker and a test fake, you wanted a protocol.

Case 3: an incomplete ABC fails at construction

Save as `abc_incomplete.py`. Forgetting `send` is a `TypeError` when you call `Broken()`, not later when you call `send`.

```
# abc_incomplete.py
from abc import ABC, abstractmethod


class TicketSink(ABC):
    @abstractmethod
    def send(self, ticket_id: int, message: str) -> None:
        raise NotImplementedError


class Broken(TicketSink):
    pass


def main() -> None:
    try:
        Broken()
    except TypeError as e:
        print(e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python abc_incomplete.py
```

Output:

```
Can't instantiate abstract class Broken without an implementation for abstract method 'send'
```

That is the ABC's best feature: a missing method is loud. A protocol cannot do this at runtime without extra tools.

Case 4: `isinstance` on a protocol, if you must

Save as `runtime_sink.py`. `@runtime_checkable` makes `isinstance` look at **method names**, not the MRO. It does not check signatures.

```
# runtime_sink.py
from typing import Protocol, runtime_checkable


@runtime_checkable
class TicketSink(Protocol):
    def send(self, ticket_id: int, message: str) -> None: ...


class Printer:
    def send(self, ticket_id: int, message: str) -> None:
        print(f"#{ticket_id} fired")


def main() -> None:
    p = Printer()
    print(isinstance(p, TicketSink))
    p.send(7, "fired")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python runtime_sink.py
```

Output:

```
True
#7 fired
```

Use this sparingly. A function that takes `TicketSink` already told the checker. Runtime `isinstance` against a protocol is a sniff test, not a proof.

The trap

An ABC hierarchy so every sink shares `log`, `retry`, `format`, and `send`. Tests now need a stub subclass. A protocol with those four methods is the same trap in structural clothing.

Keep `TicketSink` at `send`. Logging belongs in the printer, or in a wrapper function. If two sinks share a chunk of real code, a function is enough. An ABC with a concrete helper is fine when the helper is small and the subclasses are yours.

The other trap: using an ABC because a tutorial started with `class Animal(ABC)`. A desk ticket is not a taxonomy.

The boring rule

- Default seam: a `Protocol` with one or two methods.
- Tests: a real tiny class with those methods, not a mock of a concrete printer.
- ABC when you own the subclasses and you want construction to fail if a method is missing.
- `@runtime_checkable` only when some code path truly needs `isinstance`.
- Do not require inheritance so a checker will accept a fake.
- Do not put twelve methods on either kind of interface.

Try this

1. Add a `FileSink` class to `ticket_sink_protocol.py` that appends a line to a list named `lines` (same as `MemorySink`). Call `fire` on it. You do not need a new protocol.
2. Make `NotASink` subclass `TicketSink` in `ticket_sink_abc.py` and confirm `isinstance` becomes `True`.
3. Add a second abstract method `close(self) -> None` to the ABC. See `Printer()` fail until you implement `close`.
4. Drop `@runtime_checkable` from `runtime_sink.py` and wrap `isinstance` in `try/except TypeError`. Print the exception — instance checks need the decorator.

Generics and Type Parameters

Generics and Type Parameters

A **generic** is a function or class that keeps an element type through the call. `first` on a list of tables returns a table, not `object`. In 3.14 you write that with a **type parameter list**: `def first[T](items: list[T]) -> T | None`. The boring default is to genericize **one** helper that is truly reused. A function that only ever sees ticket ids stays `int`.

Mental model

`T` is a placeholder. At a call site the checker **binds** it:

- `first([12, 4])` → `T` is `int` → return `int | None`
- `first(["open", "mid"])` → `T` is `str` → return `str | None`

You do not pass `T` at runtime. `first([12, 4])` is ordinary Python.

PEP 695 (the `[T]` syntax) is the 3.12+ spelling. `TypeVar` is the older one:

```
from typing import TypeVar

T = TypeVar("T")

def first(items: list[T]) -> T | None:
    ...
```

Use `TypeVar` when you need a bound or a constraint the new syntax does not cover in your checker, or when you are editing a file that already uses it. New desk code: `[T]`.

Do not genericize because it looks like a library. If every caller passes `list[int]`, the signature is `list[int]`.

Worked examples

Case 1: `first[T]`

Save as `first_item.py`. Empty list → `None`. Otherwise the first element, same type as the list.

```
# first_item.py
def first[T](items: list[T]) -> T | None:
    if not items:
        return None
```

```
    return items[0]


def main() -> None:
    print(first([12, 4, 8]))
    print(first(["open", "mid"]))
    print(first([]))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python first_item.py
```

Output:

```
12
open
None
```

`first([])` has nothing to bind `T` from. The return is `None`. Checkers may infer `T` as `Never` or `Unknown` for that call; you still handle `None`.

Case 2: a small generic class

Save as `desk_stack.py`. The stack stores one kind of thing. `DeskStack[int]` is a stack of table numbers.

```
# desk_stack.py
class DeskStack[T]:
    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T | None:
        if not self._items:
            return None
        return self._items.pop()


def main() -> None:
    tables = DeskStack[int]()
```

```
    tables.push(12)
    tables.push(4)
    print(tables.pop())
    print(tables.pop())
    print(tables.pop())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_stack.py
```

Output:

```
4
12
None
```

`DeskStack[int]()` is a runtime subscript in 3.14 (it works). A checker uses it to type `push` and `pop`. `tables.push("12")` is a checker error; at runtime the list would happily store the string. Same lesson as the rest of typing: the hint is the contract, not a guard.

If you only ever stack tickets, write `DeskStack` with `int` and skip `[T]`.

Case 3: when not to genericize

Save as `ticket_title.py`. The id is an `int`. Wrapping this in `[T]` would let `title("seven")` type-check. That is a worse contract.

```
# ticket_title.py
def title(ticket_id: int) -> str:
    return f"ticket {ticket_id}"


def main() -> None:
    print(title(7))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_title.py
```

Output:

```
ticket 7
```

Ask: "will a second element type show up?" If no, keep the concrete type. Aliases (`type TicketId = int`) are for meaning. Generics are for **reuse across types**.

The trap

A generic that accepts everything and teaches nothing.

```python
# overgeneric.py
def title[T](ticket_id: T) -> str:
    return f"ticket {ticket_id}"


def main() -> None:
    print(title(7))
    print(title("seven"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python overgeneric.py
```

Output:

```
ticket 7
ticket seven
```

`T` is unused as a constraint. You built a function from `object` to `str` and gave it a type parameter as decoration. Write `ticket_id: int`.

The other trap: mixing `TypeVar` and `[T]` in one module for the same idea. Pick PEP 695. Reach for `TypeVar` when you need `TypeVar("T", bound=SomeClass)` or `TypeVar("T", int, str)` and your checker still wants that form — not as a default.

Do not invent `DeskContainer[T]` for a list. `list[T]` already exists.

The boring rule

- New generics: `def first[T](...)` and `class DeskStack[T]:`.
- `TypeVar` only when a bound or an old file requires it.

- Genericize a helper that is used with **more than one** element type, or a collection you will reuse.
- A function that only takes ticket ids is `int` (or `type TicketId = int`), not `[T]`.
- Empty input → `T | None` (or raise). Do not return a magic default of the wrong type.
- Do not wrap `list` in a custom generic class unless you have behavior `list` lacks (`push`/`pop` is borderline; a function on a list is simpler).

Try this

1. Add `last[T](items: list[T]) -> T | None` next to `first` and print `last([12, 4, 8])`.
2. In `desk_stack.py`, make a `DeskStack[str]` of shift names. Push `"open"` and `"mid"`, then pop twice.
3. Change `title` to take `type TicketId = int`. Keep it non-generic.
4. Try `TypeVar` on `first` in a copy of `first_item.py`. Confirm the runtime output is unchanged. Then delete the copy and keep `[T]`.

Errors

Exceptions as Control Flow

Exceptions as Control Flow

Python uses **exceptions** for failure and for some expected misses. `try` / `except` / `else` / `finally` is the full shape. The boring default is to catch **specific** types (`ValueError`, `KeyError`) and let the rest surface. Do not write `except:`. Do not catch `Exception` unless you are at a process boundary and you will log the type.

Mental model

EAFP — easier to ask forgiveness than permission — means try the operation and handle the error: `board[ticket_id]` inside `except KeyError`. **LBYL** — look before you leap — means test first: `if ticket_id not in board`.

Both are valid. EAFP is the usual Python style for dicts and files because the check and the use can race in concurrent code, and because the happy path stays unindented. LBYL is clearer when the test is a real business rule (`if n <= 0`) rather than a type of miss.

`try` runs the body.

- `except Type:` runs if that error (or a subclass) was raised.
- `else` runs if **no** exception left the `try`.
- `finally` always runs, whether you returned, raised, or succeeded.

`else` is for "the conversion worked; now use `n`." It is not required. `finally` is for cleanup when you do not have a `with`. The next chapters cover `with`.

Worked examples

Case 1: `try` / `except` / `else` / `finally`

Save as `parse_table.py`. Bad text is a `ValueError` from `int`. The `finally` line runs for both paths.

```
# parse_table.py
def parse_table(raw):
    try:
        n = int(raw)
    except ValueError:
        print("not a number")
        return
    else:
```

```
        print(f"table {n}")
    finally:
        print("done reading")


def main():
    parse_table("12")
    parse_table("twelve")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python parse_table.py
```

Output:

```
table 12
done reading
not a number
done reading
```

`return` inside `except` still runs `finally` before the function actually returns.

Case 2: EAFP on the board

Save as `board_lookup.py`. Index the dict. On `KeyError`, raise a domain error the caller can show.

```
# board_lookup.py
def table_for(ticket_id, board):
    try:
        return board[ticket_id]
    except KeyError:
        raise ValueError(f"ticket {ticket_id} not on the board")


def main():
    board = {7: 12, 8: 4}
    print(table_for(7, board))
    try:
        print(table_for(9, board))
    except ValueError as e:
        print(e)
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python board_lookup.py
```

Output:

```
12
ticket 9 not on the board
```

Catching `KeyError` and raising `ValueError` turns a builtin miss into a desk sentence. The next chapter covers `raise ... from` so the `KeyError` stays in the traceback on purpose.

Case 3: LBYL when the check *is* the rule

Save as `lbyl_board.py`. Membership is the whole story. An `if` reads better than `try` here.

```
# lbyl_board.py
def table_for(ticket_id, board):
    if ticket_id not in board:
        raise ValueError(f"ticket {ticket_id} not on the board")
    return board[ticket_id]


def main():
    board = {7: 12}
    print(table_for(7, board))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python lbyl_board.py
```

Output:

```
12
```

Use LBYL for ranges, empty strings, and "is this status paid." Use EAFP for "the dict may not have this key" and "the file may not exist."

Case 4: two types in one handler

Save as `parse_seats.py`. `int(None)` is `TypeError`. `int("four")` is `ValueError`. Both are "this raw value is not a number."

```
# parse_seats.py
def parse_seats(raw):
    try:
        n = int(raw)
    except (TypeError, ValueError) as e:
        print(f"skip: {e}")
        return None
    if n <= 0:
        raise ValueError(f"seats {n} must be positive")
    return n


def main():
    print(parse_seats("4"))
    print(parse_seats("four"))
    print(parse_seats(None))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python parse_seats.py
```

Output:

```
4
skip: invalid literal for int() with base 10: 'four'
None
skip: int() argument must be a string, a bytes-like object or a real number, not 'NoneType'
None
```

Bind the exception with `as e` when you print or wrap it. Parentheses around the types are required for a tuple of exceptions.

The trap

A bare `except:` catches **everything**, including `KeyboardInterrupt` and `SystemExit`. The process will not die when you press Ctrl-C the way you expect. It will also hide bugs.

```
# swallow.py
def main():
    raw = "twelve"
    try:
        print(int(raw))
    except:
        print("something went wrong")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python swallow.py
```

Output:

```
something went wrong
```

You lost the type and the message. The fix is a specific type, and the message.

```
# specific_except.py
def main():
    raw = "twelve"
    try:
        print(int(raw))
    except ValueError as e:
        print(f"not a table number: {e}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python specific_except.py
```

Output:

```
not a table number: invalid literal for int() with base 10: 'twelve'
```

`except Exception:` is still broad. It is acceptable at the top of a CLI if you log `repr(e)` and exit. It is not acceptable around `int(raw)` in a helper.

The boring rule

- Catch the types you can handle. Name them.
- Never `except:`. Almost never `except Exception` inside a helper.
- EAFP for dict keys and I/O. LBYL for rules you can state in English without trying the operation.
- Use `else` for "the `try` succeeded." Use `finally` for cleanup you cannot give to `with`.
- Re-raise a domain exception when the builtin one is the wrong sentence for the caller.
- Bind `as e` when the message matters. Do not bind and ignore.

Try this

1. In `parse_table.py`, parse `"0"` and, in the `else`, reject non-positive tables with `ValueError`.
2. In `board_lookup.py`, catch `ValueError` in `main` for two missing ids and keep going (print each error).
3. Change `swallow.py` to `except ValueError as e` and print `type(e).__name__`.
4. Add a path in `parse_seats.py` that calls `parse_seats("-1")` inside `try` and prints the `ValueError` for non-positive seats.

Raising, Chaining, and Groups

Raising, Chaining, and Groups

`raise` is how you fail on purpose. `raise ... from e` **chains** the cause so the traceback shows both the `KeyError` and the `ValueError` you wrapped it in. `raise ... from None` hides the cause when it is noise. An **exception group** is several errors that happened together; `except*` unpacks them. The boring default is one specific exception, chained when you wrap, grouped when a batch has more than one independent failure.

Mental model

`raise ValueError("...")` builds a traceback from this frame up.

Inside `except KeyError as e:`, `raise ValueError("...") from e` sets `__cause__`. Python prints "The above exception was the direct cause of the following exception." That is what you want when a missing shift name *is* a missing dict key.

`from None` sets `__suppress_context__`. Use it when the builtin error would confuse the reader (`KeyError: 'brunch'` after you already said `no shift named 'brunch'`).

An `ExceptionGroup(msg, [e1, e2, ...])` is one object that **contains** others. `except ValueError` does **not** match the group. `except* ValueError` matches the **inner** `ValueError`s, possibly splitting the group. Use groups when you validate several fields and want every problem, not only the first.

Worked examples

Case 1: raise a specific error

Save as `raise_plain.py`. Guard a rule. Catch it at the edge and print the message.

```
# raise_plain.py
def open_table(n):
    if n <= 0:
        raise ValueError(f"table {n}: number must be positive")
    return f"opened {n}"


def main():
    print(open_table(12))
    try:
```

```
        open_table(0)
    except ValueError as e:
        print(e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python raise_plain.py
```

Output:

```
opened 12
table 0: number must be positive
```

`raise ValueError` without a message is legal and rude. Put the bad value in the string.

Case 2: chain the cause

Save as `chain.py`. The first line of output is the successful lookup. Then the process exits with a chained traceback.

```
# chain.py
def hours_for(name):
    known = {"open": 4, "mid": 6, "close": 5}
    try:
        return known[name]
    except KeyError as e:
        raise ValueError(f"no shift named {name!r}") from e


def main():
    print(hours_for("open"))
    hours_for("brunch")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python chain.py
```

Output (the process exits non-zero):

```
4
Traceback (most recent call last):
  File "chain.py", line 5, in hours_for
    return known[name]
           ~~~~~^^^^^^
KeyError: 'brunch'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "chain.py", line 16, in <module>
    main()
    ~~~~^^
  File "chain.py", line 12, in main
    hours_for("brunch")
    ~~~~~~~~~^^^^^^^^^^
  File "chain.py", line 7, in hours_for
    raise ValueError(f"no shift named {name!r}") from e
ValueError: no shift named 'brunch'
```

Read from the top: the `KeyError`, then the `ValueError` you meant the caller to see. Line numbers follow the file you saved.

Case 3: hide a cause that adds nothing

Save as `chain_none.py`. Same wrap, no inner traceback.

```python
# chain_none.py
def hours_for(name):
    known = {"open": 4, "mid": 6}
    try:
        return known[name]
    except KeyError:
        raise ValueError(f"no shift named {name!r}") from None


def main():
    hours_for("brunch")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python chain_none.py
```

Output (the process exits non-zero):

```
Traceback (most recent call last):
  File "chain_none.py", line 15, in <module>
    main()
    ~~~~^^
  File "chain_none.py", line 11, in main
    hours_for("brunch")
    ~~~~~~~~~^^^^^^^^^^
  File "chain_none.py", line 7, in hours_for
    raise ValueError(f"no shift named {name!r}") from None
ValueError: no shift named 'brunch'
```

Use `from None` when the inner type is an implementation detail. Use `from e` when a log of the original error would help.

Case 4: a group and `except*`

Save as `check_order.py`. Collect every field error. Raise one group. `except*` walks the `ValueError`s.

```
# check_order.py
def check_order(table, cents, seats):
    errors = []
    if table <= 0:
        errors.append(ValueError(f"table {table}"))
    if cents < 0:
        errors.append(ValueError(f"cents {cents}"))
    if seats <= 0:
        errors.append(ValueError(f"seats {seats}"))
    if errors:
        raise ExceptionGroup("bad order", errors)
    return "ok"


def main():
    print(check_order(12, 850, 4))
    try:
        check_order(0, -10, 2)
    except* ValueError as group:
        for e in group.exceptions:
            print(e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python check_order.py
```

Output:

```
ok
table 0
cents -10
```

`seats` was 2, so it is not in the group. Two problems, one raise, both printed. `except ValueError` would miss the group. `except ExceptionGroup` would catch the wrapper; `except*` is the handler that talks about the inner types.

If a group also contained a `TypeError`, `except* ValueError` would handle the `ValueError`s and re-raise a group of what remains.

The trap

`raise e` after `except ValueError as e` (bare re-raise of a caught instance) resets the traceback in surprising ways. To re-raise the same error, write `raise` with nothing after it.

Wrapping without `from` still attaches an implicit context (`__context__`) and prints "During handling of the above exception, another exception occurred." That sentence is for a bug in the `except` block, not for a wrap you meant. If you wrap, say `from e` or `from None` on purpose.

Building an `ExceptionGroup` for a single error is theatre. `raise errors[0]` if the list has one item, or always group — pick one style for the validator and keep it.

The boring rule

- Raise a specific type with the bad value in the message.
- When wrapping, use `from e` or `from None`. Do not leave the default context by accident.
- `raise` with no operand re-raises the active exception.
- Validate a batch with `ExceptionGroup` and handle with `except*`.
- Do not use `except*` for ordinary single exceptions.
- Do not raise strings. Do not raise `BaseException` subclasses you do not own.

Try this

1. In `raise_plain.py`, raise `ValueError` for `n > 50` as well ("too many tables").
2. In `chain.py`, catch `ValueError` in `main` and print `e.__cause__`.
3. Switch `chain.py` to `from None` and compare the traceback to `chain_none.py`.
4. Call `check_order(0, -10, 0)` so all three fields fail. Print how many items are in `group.exceptions`.

Context Managers

Context Managers

A **context manager** is an object you use with `with`. It sets something up, yields a value, and tears it down even if the block raises. Files, locks, and "this table is occupied" are the boring uses. The boring default is `with` for every resource that has a close, and `contextlib.contextmanager` for a tiny custom one. A class with `__enter__` / `__exit__` is fine when the state has a name.

Mental model

```
with manager as name:
    ...
```

Python calls `manager.__enter__()` and binds the return value to `name`. When the block leaves — return, exception, or the last line — it calls `manager.__exit__(exc_type, exc, tb)`.

If `__exit__` returns a true value, the exception is **swallowed**. Return `False` (or `None`) to let it propagate. Swallowing is a defect unless you are writing a helper whose job is to ignore a specific error.

`@contextmanager` turns a generator into the same protocol: code before `yield` is enter, code in `finally` after `yield` is exit.

`threading.Lock` is a context manager. `with lock:` acquires and releases. Do not call `acquire()`/`release()` by hand unless you are in a situation `with` cannot express.

Worked examples

Case 1: a file in a temporary directory

Save as `shift_file.py`. `TemporaryDirectory` and `path.open()` are both context managers. Nothing is left on disk when `main` returns.

```
# shift_file.py
from pathlib import Path
import tempfile


def main():
    with tempfile.TemporaryDirectory() as d:
        path = Path(d) / "shift.txt"
```

```
        path.write_text("open 4\n")
        with path.open() as f:
            print(f.read().strip())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_file.py
```

Output:

```
open 4
```

`path.write_text` closes the file itself. The inner `with path.open()` is the read side. Nested `with` is normal.

Case 2: `__enter__` and `__exit__`

Save as `occupied.py`. Seating a table is a pair of actions. The class holds the table number.

```
# occupied.py
class Occupied:
    def __init__(self, table):
        self.table = table

    def __enter__(self):
        print(f"seat {self.table}")
        return self.table

    def __exit__(self, exc_type, exc, tb):
        print(f"clear {self.table}")
        return False


def main():
    with Occupied(12) as table:
        print(f"serving {table}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python occupied.py
```

Output:

```
seat 12
serving 12
clear 12
```

`__exit__` still runs if the body raises. Save as `occupied_error.py` to see the order.

```
# occupied_error.py
class Occupied:
    def __init__(self, table):
        self.table = table

    def __enter__(self):
        print(f"seat {self.table}")
        return self.table

    def __exit__(self, exc_type, exc, tb):
        print(f"clear {self.table}")
        return False


def main():
    try:
        with Occupied(12):
            raise ValueError("kitchen closed")
    except ValueError as e:
        print(e)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python occupied_error.py
```

Output:

```
seat 12
clear 12
kitchen closed
```

Clear, then the exception. That is the contract.

Case 3: `@contextmanager`

Save as `occupied_cm.py`. Same seating, less class. Use this when there is no extra state beyond the arguments.

```
# occupied_cm.py
from contextlib import contextmanager


@contextmanager
def occupied(table):
    print(f"seat {table}")
    try:
        yield table
    finally:
        print(f"clear {table}")


def main():
    with occupied(12) as table:
        print(f"serving {table}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python occupied_cm.py
```

Output:

```
seat 12
serving 12
clear 12
```

The `try` / `finally` around `yield` is required so `clear` runs on error. Forgetting it is the generator version of swallowing cleanup.

Case 4: a lock

Save as `desk_lock.py`. The counter is shared. The lock makes `stamp` safe if two threads called it. One thread still uses `with lock` so the acquire/release pairing cannot drift.

```
# desk_lock.py
import threading
```

```
lock = threading.Lock()
next_id = 0


def stamp():
    global next_id
    with lock:
        next_id += 1
        return f"ticket {next_id}"


def main():
    print(stamp())
    print(stamp())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_lock.py
```

Output:

```
ticket 1
ticket 2
```

`global` is ugly. A small class with `self.next_id` and `self.lock` is the next step when this grows. The `with lock` stays.

The trap

`__exit__` returning `True` eats the error. The `with` looks successful.

```
# swallow_exit.py
class Occupied:
    def __init__(self, table):
        self.table = table

    def __enter__(self):
        print(f"seat {self.table}")
        return self.table

    def __exit__(self, exc_type, exc, tb):
        print(f"clear {self.table}")
```

```
        return True


def main():
    with Occupied(12):
        raise ValueError("kitchen closed")
    print("kept going")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python swallow_exit.py
```

Output:

```
seat 12
clear 12
kept going
```

The kitchen closed and `main` continued. Return `False`. If you meant to handle `ValueError`, catch it **inside** the block or outside the `with`, not by a boolean from `__exit__`.

The boring rule

- `with` for files, locks, temp directories, and any pair of setup/teardown.
- Class with `__enter__` / `__exit__` when the manager has state. `@contextmanager` when it is a short generator.
- `__exit__` returns `False`. Cleanup goes there; handling goes in `except`.
- Put `yield` inside `try` / `finally` in a `@contextmanager` function.
- Nest `with` or use `with a, b:` for two managers. `ExitStack` is the next chapter.
- Do not open a file in `__init__` and hope the caller closes it.

Try this

1. In `shift_file.py`, write two lines and print `f.readline().strip()` twice inside the `with`.
2. Change `occupied.py` so `__enter__` returns `self`. Print `ctx.table` inside the block.
3. In `occupied_cm.py`, raise `ValueError("kitchen closed")` inside the `with` and catch it in `main`. Confirm `clear` still prints.
4. Return `False` from `swallow_exit.py`'s `__exit__` and catch the `ValueError` in `main`.

Resource Cleanup

Resource Cleanup

Every open file, lock, and temp directory needs a matching close. `try` / `finally` is the primitive. `with` is the default. `ExitStack` is the tool when the number of managers is not known until runtime. The boring default is `with`, plus `tempfile.TemporaryDirectory` so examples (and tests) leave **no** files behind.

Mental model

If you acquire it, you release it on **every** path: success, `return`, and exception.

`try` / `finally` is explicit:

```
f = path.open("w")
try:
    f.write(...)
finally:
    f.close()
```

`with path.open("w") as f:` is the same pairing, shorter, and harder to skip. Prefer it.

`contextlib.ExitStack` is a stack of context managers. `enter_context(cm)` calls `__enter__` now and schedules `__exit__` for when the stack unwinds. `callback(fn)` schedules a function. Use it for "open N shift files" where `N` comes from a list.

`tempfile.TemporaryDirectory` and `NamedTemporaryFile` create paths that vanish when the `with` ends. Do not write tutorial files next to the script.

Worked examples

Case 1: `try` / `finally`

Save as `try_finally_file.py`. The close happens even if `write` raised. We still use a temp directory so the path is not a leftover.

```
# try_finally_file.py
from pathlib import Path
import tempfile
```

```
def main():
    with tempfile.TemporaryDirectory() as d:
        path = Path(d) / "shift.txt"
        f = path.open("w")
        try:
            f.write("open 4\n")
        finally:
            f.close()
        print(path.read_text().strip())
        print(f.closed)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python try_finally_file.py
```

Output:

```
open 4
True
```

This is what you write when the object is **not** a context manager. For a file, it is the long spelling of `with`.

Case 2: the same job with `with`

Save as `with_file.py`. One block, close is implied.

```
# with_file.py
from pathlib import Path
import tempfile


def main():
    with tempfile.TemporaryDirectory() as d:
        path = Path(d) / "shift.txt"
        with path.open("w") as f:
            f.write("open 4\n")
        print(path.read_text().strip())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python with_file.py
```

Output:

```
open 4
```

After the inner `with`, `f` is closed. `path.read_text()` opens and closes its own handle. Nested `with` on the directory plus the file is the usual shape.

Case 3: `ExitStack` for a dynamic set of files

Save as `exit_stack.py`. Three shift files, one stack. All three close when the stack's `with` ends — even if the second `write` failed.

```
# exit_stack.py
from contextlib import ExitStack
from pathlib import Path
import tempfile


def main():
    with tempfile.TemporaryDirectory() as d:
        root = Path(d)
        names = ("open", "mid", "close")
        with ExitStack() as stack:
            files = []
            for name in names:
                path = root / f"{name}.txt"
                f = stack.enter_context(path.open("w"))
                f.write(f"{name} shift\n")
                files.append(path.name)
            print(len(files))
        print((root / "mid.txt").read_text().strip())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python exit_stack.py
```

Output:

```
3
mid shift
```

Do not nest `with open(...)` three levels deep because you happen to have three names. Do not keep a list of handles and close them in a loop you might not reach.

Case 4: a callback when there is no context manager

Save as `exit_callback.py`. `stack.callback(f.close)` is `finally: f.close()` on the stack.

```
# exit_callback.py
from contextlib import ExitStack
from pathlib import Path
import tempfile


def main():
    with tempfile.TemporaryDirectory() as d:
        path = Path(d) / "shift.txt"
        with ExitStack() as stack:
            f = path.open("w")
            stack.callback(f.close)
            f.write("open 4\n")
        print(f.closed)
        print(path.read_text().strip())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python exit_callback.py
```

Output:

```
True
open 4
```

Prefer `enter_context(path.open("w"))` for files. Use `callback` for a close function you already have.

A helper that only needs one path can stay tiny:

```
# named_temp.py
from pathlib import Path
import tempfile
```

```
def write_shift(path, name, hours):
    path.write_text(f"{name} {hours}\n")
    return path.read_text().strip()


def main():
    with tempfile.TemporaryDirectory() as d:
        path = Path(d) / "shift.txt"
        print(write_shift(path, "open", 4))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python named_temp.py
```

Output:

```
open 4
```

`write_text` / `read_text` close internally. The directory still needs the `with` so the folder goes away.

The trap

Opening files in a loop and hoping CPython closes them when the function returns — `handles = [path.open("w") for path in paths]` — is a leak. The list holds the handles alive. An exception before you close them leaves them open. On some systems you run out of file descriptors before you run out of patience.

The other trap is a hard-coded path (`/tmp/shift.txt`) in a book example or a test. The next run collides. A crashed run leaves the file. `TemporaryDirectory` is the whole fix.

`try` / `finally` that calls `close` but forgets it on the success path (close only in `except`) is the same leak. `finally` or `with`, not `except` alone.

The boring rule

- Prefer `with` over `try` / `finally` when the object supports it.
- Use `try` / `finally` for acquire/release pairs that are not context managers.
- Use `ExitStack` when the count of managers is dynamic.

- Use `tempfile.TemporaryDirectory` (or `NamedTemporaryFile`) in programs and tests that need a path. Do not leave files in the working directory.
- Close in reverse order of open. `ExitStack` does that.
- Do not rely on garbage collection to close files or sockets.

Try this

1. In `try_finally_file.py`, raise `ValueError("full")` after `write`, catch it outside the `try` / `finally`, and print `f.closed` — it should still be `True`.
2. Rewrite `exit_callback.py` to use `enter_context` instead of `callback`. Keep the prints.
3. Add a fourth name `"bar"` to the `names` tuple in `exit_stack.py`. Confirm `len(files)` is `4`.
4. In `named_temp.py`, write two files (`open.txt`, `mid.txt`) in the same temp directory and print both.

Iteration

Iterators

Iterators

A `for` loop in Python is not special syntax over indexes. It asks an object for an **iterator**, then calls `next` until the iterator is empty. The boring default is: use `for` on a list, tuple, or dict. Learn `iter` / `next` so you can write a stream when a full list would be waste, and so you stop being surprised when a stream is empty the second time you walk it.

Mental model

An **iterable** is anything `iter(x)` accepts: a list of tickets, a dict of tables, a file, a custom class with `__iter__`. That call returns an **iterator**.

An iterator is a stateful cursor. It has `__next__`. Each call yields one item. When there is nothing left it raises `StopIteration`. A `for` loop catches that exception and ends. You almost never catch `StopIteration` yourself.

`iter(x)` on an iterator usually returns the same object. `iter(x)` on a list returns a **new** cursor each time. That is why you can loop a list twice and a generator once.

Worked examples

Case 1: `iter` and `next`

Save as `iter_next.py`. Three tickets sit on the rail. `iter` hands you a cursor; `next` pulls one ticket at a time.

```
# iter_next.py
def main():
    tickets = ["T-11", "T-12", "T-13"]
    cursor = iter(tickets)
    print(type(cursor).__name__)
    print(next(cursor))
    print(next(cursor))
    print(next(cursor))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python iter_next.py
```

Output:

```
list_iterator
T-11
T-12
T-13
```

The list is still intact. The cursor moved. `tickets` is the iterable; `cursor` is the iterator.

Case 2: `StopIteration` and a default

Save as `next_default.py`. A fourth `next` has nothing to return. Bare `next` raises. `next(cursor, default)` does not.

```
# next_default.py
def main():
    cursor = iter(["T-11"])
    print(next(cursor))
    print(next(cursor, "window closed"))
    cursor = iter([])
    try:
        next(cursor)
    except StopIteration:
        print("empty window")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python next_default.py
```

Output:

```
T-11
window closed
empty window
```

Use the default form at a boundary (a queue that may be empty). Do not wrap every `next` in `try`. A `for` loop already does that job.

Case 3: What `for` actually does

Save as `for_protocol.py`. The loop on the left is the protocol on the right. Same tickets, same order.

```
# for_protocol.py
def walk(tickets):
    cursor = iter(tickets)
    while True:
        try:
            ticket = next(cursor)
        except StopIteration:
            break
        print(f"called {ticket}")


def main():
    tickets = ["T-11", "T-12"]
    for ticket in tickets:
        print(f"for {ticket}")
    walk(tickets)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python for_protocol.py
```

Output:

```
for T-11
for T-12
called T-11
called T-12
```

Write the `while` version only when you are teaching the protocol, or when you need `next` with a sentinel. At the desk, write `for`.

Case 4: A custom iterator class

Save as `ticket_window.py`. The window holds a list and a position. `__iter__` returns `self`, so the object **is** its own iterator. `__next__` raises `StopIteration` when the rail is empty.

```python
# ticket_window.py
class TicketWindow:
    def __init__(self, tickets):
        self._tickets = list(tickets)
        self._i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._i >= len(self._tickets):
            raise StopIteration
        ticket = self._tickets[self._i]
        self._i += 1
        return ticket


def main():
    window = TicketWindow(["T-21", "T-22", "T-23"])
    print(next(window))
    for ticket in window:
        print(f"served {ticket}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_window.py
```

Output:

```
T-21
served T-22
served T-23
```

`for ticket in window` did not restart. `next` had already taken `T-21`. The class is a cursor, not a catalog.

The trap

Save as `twice.py`. A list can be walked twice because each `for` calls `iter` and gets a fresh cursor. A `TicketWindow` returns itself, so the second loop is already at the end.

```
# twice.py
class TicketWindow:
    def __init__(self, tickets):
        self._tickets = list(tickets)
        self._i = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._i >= len(self._tickets):
            raise StopIteration
        ticket = self._tickets[self._i]
        self._i += 1
        return ticket


def main():
    rail = ["T-31", "T-32"]
    print("list", list(rail), list(rail))
    window = TicketWindow(rail)
    print("window", list(window), list(window))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python twice.py
```

Output:

```
list ['T-31', 'T-32'] ['T-31', 'T-32']
window ['T-31', 'T-32'] []
```

The empty second walk is not a bug in `list`. The iterator was spent. If you need two independent walks, either keep the source list, or make `__iter__` return a **new** cursor (or a generator, next chapter).

The boring rule

- Prefer `for item in tickets` over indexes and over hand-written `next` loops.
- `iter(x)` gives a cursor. `next(cursor)` pulls one item. `StopIteration` means empty.
- `next(cursor, default)` is for a boundary that may be empty.

- A custom class with `__iter__` returning `self` is a one-shot cursor. Document that, or do not write the class.
- Do not catch `StopIteration` in ordinary desk code. Let `for` do it.

Try this

1. In `iter_next.py`, print `list(cursor)` after the three `next` calls. Then print `list(iter(tickets))`.
2. In `ticket_window.py`, add a `peek` method that returns the next ticket without advancing, or `None` at the end.
3. Change `TicketWindow.__iter__` so it returns a fresh `TicketWindow(self._tickets)` instead of `self`. Run `twice.py` again.
4. Pass a dict of `{table: ticket}` to `for`. Print both the key and the value. Then iterate `.values()`.

Generators and yield

Generators and yield

A **generator** is an iterator you write as a function. `yield` pauses the function and hands a value to the caller. The next `next` (or the next step of a `for`) resumes just after that `yield`. The boring default is a generator function, not a custom iterator class. You get `__iter__` and `__next__` for free, and you do not manage an index.

Mental model

`return` ends a function. `yield` parks it. The object you get from calling a generator function is the generator — calling the function does not run the body yet. The body runs up to the first `yield` when something pulls a value.

`yield from xs` is "walk this iterable and yield each item." It is the readable way to flatten two shifts into one stream.

A **generator expression** is a comprehension in parentheses: `(t["id"] for t in tickets if t["open"])`. It is lazy. It is also one-shot, like any generator.

`send` pushes a value *into* a generator at a `yield` expression. You will rarely need it at a desk. A function argument is clearer. The example below is enough to recognise it.

Worked examples

Case 1: `yield` is a window

Save as `window_gen.py`. Compare this to the `TicketWindow` class in the previous chapter. Same behaviour, no index.

```
# window_gen.py
def ticket_window(tickets):
    for ticket in tickets:
        yield ticket


def main():
    window = ticket_window(["T-41", "T-42", "T-43"])
    print(type(window).__name__)
    print(next(window))
    for ticket in window:
```

```
        print(f"served {ticket}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python window_gen.py
```

Output:

```
generator
T-41
served T-42
served T-43
```

Calling `ticket_window(...)` built the generator. `next` ran the body until the first `yield`. The `for` continued from there.

Case 2: `yield from` merges shifts

Save as `merge_shifts.py`. Morning and evening are two lists. The desk wants one stream.

```
# merge_shifts.py
def all_tickets(morning, evening):
    yield from morning
    yield from evening


def main():
    morning = ["T-51", "T-52"]
    evening = ["T-61"]
    print(list(all_tickets(morning, evening)))
    for ticket in all_tickets(morning, evening):
        print(ticket)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python merge_shifts.py
```

Output:

```
['T-51', 'T-52', 'T-61']
T-51
T-52
T-61
```

Each call to `all_tickets` is a new generator, so `list(...)` and the `for` both see every ticket. `yield from morning` is not a copy of the list; it walks `morning` as the caller pulls.

Case 3: A generator expression

Save as `open_ids.py`. Only open tickets leave the rail. The expression does not build a second list of dicts.

```
# open_ids.py
def main():
    tickets = [
        {"id": "T-71", "open": True},
        {"id": "T-72", "open": False},
        {"id": "T-73", "open": True},
    ]
    open_ids = (t["id"] for t in tickets if t["open"])
    print(type(open_ids).__name__)
    print(list(open_ids))
    print(list(open_ids))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python open_ids.py
```

Output:

```
generator
['T-71', 'T-73']
[]
```

The second `list` is empty because the generator was spent. If you need the ids twice, use a list comprehension, or call a generator function twice.

Case 4: `send`, briefly

Save as `pager.py`. The first `next` (or `send(None)`) runs to the first `yield` and gets `"ready"`. Later `send` values appear as the result of that `yield`. Sending `None` ends the loop.

```
# pager.py
def pager():
    ticket = yield "ready"
    while ticket is not None:
        ticket = yield f"paged {ticket}"


def main():
    station = pager()
    print(next(station))
    print(station.send("T-81"))
    print(station.send("T-82"))
    try:
        station.send(None)
    except StopIteration:
        print("pager closed")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python pager.py
```

Output:

```
ready
paged T-81
paged T-82
pager closed
```

If you find yourself designing a protocol around `send`, stop. Pass the ticket as a function argument, or put tickets on a queue. `send` is in the language; it is not the desk default.

The trap

Save as `class_vs_gen.py`. The class stores an index and a copy of the list. The generator does not. Prefer the generator unless you need extra methods (`peek`, `close_window`) that a function cannot hold.

```
# class_vs_gen.py
class TicketWindow:
    def __init__(self, tickets):
        self._tickets = list(tickets)
        self._i = 0
```

```
    def __iter__(self):
        return self

    def __next__(self):
        if self._i >= len(self._tickets):
            raise StopIteration
        ticket = self._tickets[self._i]
        self._i += 1
        return ticket


def ticket_window(tickets):
    for ticket in tickets:
        yield ticket


def main():
    rail = ["T-91", "T-92"]
    print("class", list(TicketWindow(rail)))
    print("gen", list(ticket_window(rail)))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python class_vs_gen.py
```

Output:

```
class ['T-91', 'T-92']
gen ['T-91', 'T-92']
```

Same output. The generator is the one you keep. Write a class when the window has state a caller must poke between pulls. That is rarer than tutorials suggest.

The boring rule

- Write a generator function with `yield` instead of an iterator class.
- `yield from` flattens another iterable. Nested `for` plus `yield` is the long form of the same idea.
- Generator expressions are lazy and one-shot. List comprehensions are eager and reusable.
- Do not build a `send` protocol. Call a function.
- `list(gen)` is fine for a small rail. Do not `list()` a stream you meant to keep lazy.

Try this

1. In `window_gen.py`, `yield` a formatted string `served {ticket}` instead of the raw id.
2. Add a third iterable `late` to `all_tickets` and `yield from` it last.
3. In `open_ids.py`, replace the generator expression with a generator function `def open_ids(tickets):` that `yield`s. Call it twice.
4. Drop `send` from `pager.py`. Make `pager(tickets)` a generator that yields `paged {ticket}` for each id in a list argument.

itertools and collections

itertools and collections

The standard library already walks, groups, counts, and queues. `itertools` builds iterators. `collections` builds specialised containers. The boring default is these two modules, not a new class, when the job is "number tickets," "stick two shifts together," "group by table," "tally statuses," or "serve from the front of the line."

Mental model

`itertools` functions return iterators. They do not copy the source unless the algorithm has to. `count` never ends on its own — bound it. `chain` walks one iterable after another. `groupby` walks **consecutive** equal keys, so sort first if you want every table together.

`collections.Counter` counts hashable things. `collections.deque` is a list that pops from the left in constant time. `collections.defaultdict` calls a factory for missing keys so you do not write `if key not in d`.

No extra packages. If you need something these do not do, write a short function.

Worked examples

Case 1: `count` numbers the rail

Save as `ticket_ids.py`. The desk issues ids starting at 100. `zip` stops when the names stop, so `count` does not run forever.

```
# ticket_ids.py
from itertools import count


def main():
    names = ["soup", "grill", "bar"]
    for number, name in zip(count(100), names):
        print(f"T-{number} {name}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_ids.py
```

Output:

```
T-100 soup
T-101 grill
T-102 bar
```

`count(100, 5)` would step by five. A `for n in count(100):` without a `break` or a `zip` is a live lock. Bound the work.

Case 2: `chain` joins shifts

Save as `chain_shifts.py`. Morning and evening stay as two lists. The loop sees one stream.

```
# chain_shifts.py
from itertools import chain


def main():
    morning = ["T-11", "T-12"]
    evening = ["T-21"]
    for ticket in chain(morning, evening):
        print(ticket)
    print(list(chain.from_iterable([morning, evening])))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python chain_shifts.py
```

Output:

```
T-11
T-12
T-21
['T-11', 'T-12', 'T-21']
```

`chain.from_iterable` is for a list of lists. `chain(morning, evening)` is for a handful of named iterables. Do not concatenate with `morning + evening` if you only need to walk them.

Case 3: `groupby` needs a sort

Save as `by_table.py`. Tickets arrive mixed. `groupby` only clusters **neighbours** with the same key. Sort by table first.

```
# by_table.py
from itertools import groupby


def main():
    tickets = [
        {"id": "T-1", "table": 4},
        {"id": "T-2", "table": 2},
        {"id": "T-3", "table": 4},
        {"id": "T-4", "table": 2},
    ]
    ordered = sorted(tickets, key=lambda t: t["table"])
    for table, group in groupby(ordered, key=lambda t: t["table"]):
        ids = [t["id"] for t in group]
        print(f"table {table}: {ids}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python by_table.py
```

Output:

```
table 2: ['T-2', 'T-4']
table 4: ['T-1', 'T-3']
```

Consume `group` before the next iteration. It is an iterator over the current run, not a list you can keep. If you skip the sort, table 4 appears twice.

Case 4: `Counter` tallies statuses

Save as `tally.py`. A night of tickets, then a count.

```
# tally.py
from collections import Counter


def main():
    statuses = ["open", "paid", "open", "void", "paid", "open"]
```

```
    counts = Counter(statuses)
    print(counts["open"])
    print(counts.most_common(2))
    counts.update(["paid", "paid"])
    print(dict(counts))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python tally.py
```

Output:

```
3
[('open', 3), ('paid', 2)]
{'open': 3, 'paid': 4, 'void': 1}
```

`Counter` is a dict. Missing keys read as `0`, not `KeyError`. `most_common` is the report the close-of-shift wants.

Case 5: `deque` and `defaultdict`

Save as `line_and_tabs.py`. The line is served from the left. Tabs accumulate by table without an `if table not in tabs` block.

```
# line_and_tabs.py
from collections import defaultdict, deque


def main():
    line = deque(["T-11", "T-12", "T-13"])
    line.append("T-14")
    print(line.popleft())
    print(list(line))

    tabs = defaultdict(int)
    for table, pence in [(4, 1200), (2, 800), (4, 400)]:
        tabs[table] += pence
    print(dict(tabs))
    print(tabs[9])


if __name__ == "__main__":
```

```
    main()
```

Run:

```
uv run python line_and_tabs.py
```

Output:

```
T-11
['T-12', 'T-13', 'T-14']
{4: 1600, 2: 800}
0
```

`tabs[9]` created a `0`. That is the deal with `defaultdict`. If a missing table should be an error, use a plain dict.

`deque(maxlen=3)` drops from the left when you append past the bound. Use that for a short recent-ticket log, not as a clever database.

The trap

Save as `groupby_unsorted.py`. Same tickets as Case 3, no sort. `groupby` reports table 4, then 2, then 4, then 2.

```
# groupby_unsorted.py
from itertools import groupby


def main():
    tickets = [
        {"id": "T-1", "table": 4},
        {"id": "T-2", "table": 2},
        {"id": "T-3", "table": 4},
        {"id": "T-4", "table": 2},
    ]
    for table, group in groupby(tickets, key=lambda t: t["table"]):
        ids = [t["id"] for t in group]
        print(f"table {table}: {ids}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python groupby_unsorted.py
```

Output:

```
table 4: ['T-1']
table 2: ['T-2']
table 4: ['T-3']
table 2: ['T-4']
```

That is consecutive grouping, not a merge. Sort, or use `defaultdict(list)` and append.

The boring rule

- `count` must be bounded (`zip`, `islice`, `break`).
- `chain` (or `yield from`) instead of building a combined list you do not need.
- Sort before `groupby`. Treat each group as a one-shot iterator.
- `Counter` for tallies. `deque` for a line. `defaultdict` when missing keys have an obvious factory.
- Stay in the standard library until a profiler or a real gap says otherwise.

Try this

1. In `ticket_ids.py`, start at 500 and print only two tickets (`zip` with a slice of `names`).
2. Replace `chain` in `chain_shifts.py` with a generator that `yield from`s both lists. Same output.
3. Rewrite `by_table.py` with `defaultdict(list)` and no sort. Print `dict(tabs)`.
4. Give the `deque` a `maxlen=3`, append five tickets, and print what remains.

Testing

Testing Fundamentals

Testing Fundamentals

A test is a small program that calls your code and **asserts** what should be true. The boring default in this book is **pytest**: files named `test_*.py`, plain `assert`, and `uv run pytest`. You do not need a `TestCase` class. You do not need a mock until a real dependency is in the way.

Mental model

Save the code under test as a module (`desk.py`). Save tests next to it as `test_desk.py`. pytest loads every `test_*` function, runs it, and reports failures with the assertion and the values.

`assert expression` is enough. When it fails, pytest prints both sides. `pytest.mark.parametrize` runs the same test with different inputs so you do not copy the function three times.

A `pyproject.toml` that lists pytest as a dev dependency is optional. `uv run --with pytest pytest` is enough to learn. Add the project file when the desk is a real package.

Worked examples

Case 1: Two files, one assertion

Save both files in the same directory.

```
# desk.py
def format_pence(pence):
    if pence < 0:
        raise ValueError("pence must be >= 0")
    return f"{pence / 100:.2f}"


def open_ticket(ticket_id, table):
    if table <= 0:
        raise ValueError(f"table {table}: must be positive")
    return {"id": ticket_id, "table": table, "status": "open"}
```

```
# test_desk.py
import pytest

from desk import format_pence, open_ticket
```

```
def test_format_pence_pounds():
    assert format_pence(1200) == "12.00"


def test_open_ticket_shape():
    ticket = open_ticket("T-11", 4)
    assert ticket["status"] == "open"
    assert ticket["table"] == 4


def test_open_ticket_rejects_table_zero():
    with pytest.raises(ValueError, match="table 0"):
        open_ticket("T-11", 0)
```

Run:

```
uv run --with pytest pytest test_desk.py -q
```

Output (duration varies):

```
...                                                                      [100%]
3 passed in 0.01s
```

`pytest.raises` is a context manager. The test **fails** if the call does not raise, or if the message does not match.

Case 2: Parametrize instead of copy-paste

Add this to `test_desk.py` (keep the functions from Case 1).

```
# test_desk.py
import pytest

from desk import format_pence, open_ticket


def test_open_ticket_shape():
    ticket = open_ticket("T-11", 4)
    assert ticket["status"] == "open"
    assert ticket["table"] == 4


def test_open_ticket_rejects_table_zero():
    with pytest.raises(ValueError, match="table 0"):
        open_ticket("T-11", 0)
```

```
@pytest.mark.parametrize(
    ("pence", "label"),
    [
        (0, "0.00"),
        (50, "0.50"),
        (1200, "12.00"),
        (99, "0.99"),
    ],
)
def test_format_pence(pence, label):
    assert format_pence(pence) == label
```

Run:

```
uv run --with pytest pytest test_desk.py -q
```

Output (duration varies):

```
......                                                                   [100%]
6 passed in 0.01s
```

Four rows in the table plus two other tests is six. A failing row names the arguments: `pence=99`. That is why parametrize beats three near-identical functions.

Case 3: Optional `pyproject.toml`

When the desk is a project, pin pytest once.

```
# pyproject.toml
[project]
name = "desk"
version = "0.1.0"
requires-python = ">=3.14"

[dependency-groups]
dev = ["pytest>=8"]
```

Then, from that directory:

```
uv sync --group dev
uv run pytest test_desk.py -q
```

Until you have a project, keep using `uv run --with pytest pytest test_desk.py`. Same tests.

The trap

Save as `test_silent.py` next to `desk.py`. A test without an `assert` always passes. pytest does not read your mind.

```
# test_silent.py
from desk import format_pence


def test_format_pence():
    format_pence(1200)
```

Run:

```
uv run --with pytest pytest test_silent.py -q
```

Output (duration varies):

```
.                                                                        [100%]
1 passed in 0.01s
```

The function returned `"12.00"` and nobody looked. Add `assert format_pence(1200) == "12.00"`. A green bar that never asserted is a live bug.

The boring rule

- Code in `desk.py`. Tests in `test_desk.py`. Names start with `test_`.
- `assert` the result. `pytest.raises` for the error you expect.
- Parametrize tables of inputs. Do not clone the test function.
- `uv run --with pytest pytest` until the project files exist; then `uv run pytest`.
- A test with no assert is not a test.

Try this

1. Add `test_format_pence_rejects_negative` that expects `ValueError` from `format_pence(-1)`.
2. Add a parametrize row `(100, "1.00")` and run again.
3. Make `open_ticket` reject an empty `ticket_id`. Add a test that would have failed before the check.
4. Run `uv run --with pytest pytest test_desk.py -q -k format` and confirm only the format tests run.

Fixtures, Parametrize, and Coverage

Fixtures, Parametrize, and Coverage

A **fixture** is a function pytest calls to build a value a test needs: a list of tickets, a temp directory, a clock. `tmp_path` is a fixture pytest already provides — a fresh `pathlib.Path` per test. **Coverage** is a report of which lines ran. The boring default is: small fixtures, parametrize for tables, and a coverage run before you call a change done.

Mental model

Put shared setup in `@pytest.fixture`. Tests name the fixture as an argument. pytest injects the return value. Do not call the fixture yourself.

`tmp_path` lives on disk for one test, then goes away. Use it for receipts and JSON, not for "the real desk folder."

`pytest-cov` measures lines. A number is not a goal. The missing-line report is: you wrote a branch and no test walked it.

Worked examples

Case 1: A fixture for the rail

Save both files in the same directory.

```
# desk.py
def open_ids(tickets):
    return [t["id"] for t in tickets if t["status"] == "open"]


def total_pence(tickets):
    return sum(t["pence"] for t in tickets)


def write_receipt(path, tickets):
    lines = [f"{t['id']} {t['pence']}" for t in tickets]
    path.write_text("\n".join(lines) + "\n")
```

```
# test_desk.py
import pytest
```

```
from desk import open_ids, total_pence, write_receipt


@pytest.fixture
def tickets():
    return [
        {"id": "T-11", "status": "open", "pence": 1200},
        {"id": "T-12", "status": "paid", "pence": 800},
        {"id": "T-13", "status": "open", "pence": 400},
    ]


def test_open_ids(tickets):
    assert open_ids(tickets) == ["T-11", "T-13"]


def test_total_pence(tickets):
    assert total_pence(tickets) == 2400
```

Run:

```
uv run --with pytest pytest test_desk.py -q
```

Output (duration varies):

```
..                                                                       [100%]
2 passed in 0.01s
```

Each test gets its own list. Mutating `tickets` in one test does not leak into the other, because the fixture function runs again.

Case 2: `tmp_path` for a receipt file

Add this test to `test_desk.py`.

```
# test_desk.py
import pytest

from desk import open_ids, total_pence, write_receipt


@pytest.fixture
def tickets():
    return [
        {"id": "T-11", "status": "open", "pence": 1200},
        {"id": "T-12", "status": "paid", "pence": 800},
```

```
        {"id": "T-13", "status": "open", "pence": 400},
    ]


def test_open_ids(tickets):
    assert open_ids(tickets) == ["T-11", "T-13"]


def test_total_pence(tickets):
    assert total_pence(tickets) == 2400


def test_write_receipt(tmp_path, tickets):
    path = tmp_path / "receipt.txt"
    write_receipt(path, tickets)
    text = path.read_text()
    assert "T-11 1200" in text
    assert text.endswith("\n")
```

Run:

```
uv run --with pytest pytest test_desk.py -q
```

Output (duration varies):

```
...                                                                      [100%]
3 passed in 0.01s
```

Do not write receipts next to the test file. `tmp_path` keeps the repo clean and the test isolated.

Case 3: Parametrize plus a fixture

Save as `test_status.py` next to the same `desk.py`.

```
# test_status.py
import pytest

from desk import open_ids


@pytest.fixture
def ticket():
    return {"id": "T-21", "status": "open", "pence": 100}


@pytest.mark.parametrize(
```

```
    ("status", "want"),
    [
        ("open", ["T-21"]),
        ("paid", []),
        ("void", []),
    ],
)
def test_open_ids_status(ticket, status, want):
    ticket["status"] = status
    assert open_ids([ticket]) == want
```

Run:

```
uv run --with pytest pytest test_status.py -q
```

Output (duration varies):

```
...                                                                      [100%]
3 passed in 0.01s
```

The fixture builds a ticket. Parametrize changes one field. Keep the table small enough to read on one screen.

Case 4: Coverage

From the directory that holds `desk.py` and `test_desk.py` (Case 2):

```
uv run --with pytest --with pytest-cov pytest test_desk.py --cov=desk --cov-report=term-miss
```

Output (duration and column spacing vary):

```
...                                                                      [100%]
============================== tests coverage ==============================
_______________ coverage: platform linux, python 3.14.7-final-0 _______________

Name      Stmts   Miss  Cover   Missing
-----------------------------------------
desk.py       7      0   100%
-----------------------------------------
TOTAL         7      0   100%
3 passed in 0.03s
```

`--cov=desk` is the **module** name, not the test file. `Missing` lists line numbers no test ran. Add a test for that branch; do not delete the branch to make the number pretty.

A `pyproject.toml` can pin both tools:

```
# pyproject.toml
[project]
name = "desk"
version = "0.1.0"
requires-python = ">=3.14"

[dependency-groups]
dev = ["pytest>=8", "pytest-cov>=6"]
```

Then `uv sync --group dev` and `uv run pytest test_desk.py --cov=desk --cov-report=term-missing`.

The trap

Save this `desk.py` and a test that never voids a ticket.

```
# desk.py
def close(ticket, status):
    if status not in {"paid", "void"}:
        raise ValueError(f"bad status {status!r}")
    ticket = dict(ticket)
    ticket["status"] = status
    return ticket
```

```
# test_close.py
from desk import close


def test_close_paid():
    ticket = close({"id": "T-1", "status": "open"}, "paid")
    assert ticket["status"] == "paid"
```

Run coverage:

```
uv run --with pytest --with pytest-cov pytest test_close.py --cov=desk --cov-report=term-mis
```

You will see the `ValueError` line in `Missing`. The paid path is green. The bad-status path is not. Shipping at "most of the function ran" is how voided tickets skip the check.

The boring rule

- Fixtures build data. Tests name them as arguments.
- `tmp_path` for files. Never write into the source tree from a test.
- Parametrize the inputs. Fixture the setup that is the same every time.
- Run `--cov=desk --cov-report=term-missing` and read `Missing`, not only the percent.
- 100% coverage is not a personality. Untested branches are.

Try this

1. Add a fixture `paid_only` that filters Case 1's tickets. Assert `open_ids` is empty.
2. In `test_write_receipt`, assert the file has exactly three lines of content (plus the final newline).
3. Add a `void` path test for the trap's `close` and re-run coverage until `Missing` is empty.
4. Run pytest on `test_desk.py` and `test_status.py` together: `uv run --with pytest pytest test_desk.py test_status.py -q`.

Advanced Testing

Advanced Testing

Some code talks to the environment: env vars, a printer, a clock, stdout. The boring default is still a real function with an argument you can pass in tests. When the seam is already `os.environ` or `print`, pytest's `monkeypatch` and `capsys`, and `unittest.mock.patch`, are the stdlib-shaped tools. Use them to **replace one name**. Do not mock the function you are trying to test.

Mental model

`monkeypatch` changes an attribute, env var, or dict for the length of one test, then puts it back.

`capsys` captures `print`. Read `.out` and `.err`. Prefer returning a string from the function; capture stdout when the function's job *is* to print a label.

`unittest.mock.patch("module.name")` replaces `name` where it is **used**, not where it is defined. `patch("desk.send")` if `desk.py` called `send`. The mock records calls. That is the whole trick.

Worked examples

Case 1: `monkeypatch` an env prefix

Save both files in the same directory.

```
# desk.py
import os


def ticket_id(n):
    prefix = os.environ.get("DESK_PREFIX", "T")
    return f"{prefix}-{n}"


def format_pence(pence):
    return f"{pence / 100:.2f}"
```

```
# test_desk.py
from desk import ticket_id
```

```
def test_ticket_id_default(monkeypatch):
    monkeypatch.delenv("DESK_PREFIX", raising=False)
    assert ticket_id(11) == "T-11"


def test_ticket_id_override(monkeypatch):
    monkeypatch.setenv("DESK_PREFIX", "X")
    assert ticket_id(11) == "X-11"
```

Run:

```
uv run --with pytest pytest test_desk.py -q
```

Output (duration varies):

```
..                                                                   [100%]
2 passed in 0.01s
```

`raising=False` means "delete if present." The test does not care whether you exported `DESK_PREFIX` in the shell.

Case 2: `capsys` for a printed label

```
# label.py
def print_label(ticket):
    print(f"ticket {ticket['id']} → table {ticket['table']}")
```

```
# test_label.py
from label import print_label


def test_print_label(capsys):
    print_label({"id": "T-11", "table": 4})
    captured = capsys.readouterr()
    assert captured.out == "ticket T-11 → table 4\n"
    assert captured.err == ""
```

Run:

```
uv run --with pytest pytest test_label.py -q
```

Output (duration varies):

```
.                                                                    [100%]
1 passed in 0.01s
```

The newline is part of `print`. Assert it. If you start adding `file=` and colours, return a string instead and print in `main`.

Case 3: `unittest.mock` replaces a sender

`page` should not hit a real radio in a unit test. Patch `send` where `page` looks it up: `kitchen.send`.

```
# kitchen.py
def send(message):
    raise RuntimeError(f"no radio for {message!r}")


def page(ticket):
    send(f"fire {ticket}")
    return ticket
```

```
# test_kitchen.py
from unittest.mock import patch

import kitchen


def test_page_sends_once():
    with patch("kitchen.send") as send:
        result = kitchen.page("T-11")
        send.assert_called_once_with("fire T-11")
        assert result == "T-11"


def test_page_without_patch_blows_up():
    try:
        kitchen.page("T-11")
    except RuntimeError as exc:
        assert "no radio" in str(exc)
    else:
        raise AssertionError("expected RuntimeError")
```

Run:

```
uv run --with pytest pytest test_kitchen.py -q
```

Output (duration varies):

```
..                                                                       [100%]
2 passed in 0.02s
```

`assert_called_once_with` is the contract. The second test proves the real `send` is still dangerous — the patch did not leak.

A more boring `page` would take `send` as an argument with a default. Patch is for seams you do not want to change today.

The trap

Save as `test_overmock.py`. This test mocks `ticket_id`, then asserts the mock. It never ran the formatting logic.

```
# test_overmock.py
from unittest.mock import patch

import desk


def test_ticket_id_is_a_lie():
    with patch("desk.ticket_id", return_value="nope"):
        assert desk.ticket_id(11) == "nope"
```

Run:

```
uv run --with pytest pytest test_overmock.py -q
```

Output (duration varies):

```
.                                                                        [100%]
1 passed in 0.01s
```

Green, and `DESK_PREFIX` could be on fire. Mock the neighbour (`send`, `time.time`, `path.write_text`). Call the function you claim to test.

The boring rule

- Prefer an extra argument (`send=...`, `clock=...`) over a patch.
- `monkeypatch` for env and attributes. It rolls back by itself.
- `capsys` when the function's job is printing. Otherwise return a string.
- `patch("module.name")` at the use site. Assert calls. Leave the function under test real.
- If a test would pass with the production code deleted, you over-mocked.

Try this

1. Add `monkeypatch.setenv("DESK_PREFIX", "")` and decide whether `ticket_id(11)` should be `"-11"` or rejected. Test the decision.
2. Change `print_label` to also `print` the pence on a second line. Update the `capsys` assert.
3. Rewrite `page(ticket, send=send)` so the test passes a fake `send` list-append. Drop `patch`.
4. In `test_page_sends_once`, assert `send.call_count == 1` as well as `assert_called_once_with`.

Integration Testing

Integration Testing

An integration test runs two real pieces together: a function and the filesystem, or a client and an HTTP handler. The boring default is still a process that **starts, checks, and exits**. Do not leave `serve_forever` running. Prefer `urllib` against a function when there is no socket yet. When you do need a server, bind port `0`, start a thread, request, then `shutdown`.

Mental model

Unit tests pin one function. Integration tests pin a seam you could not fake honestly: JSON on disk, an HTTP status line, a temp directory layout.

`tempfile.TemporaryDirectory` (or pytest's `tmp_path`) is the desk folder. `http.server.HTTPServer` is the socket. `urllib.request.urlopen` is the client. Join the thread. Close the server. The test process must return.

If the handler is a function, call the function. A TCP server is for "we actually speak HTTP," not for "I heard integration tests need ports."

Worked examples

Case 1: `urllib` against a builder — no socket

Save as `ticket_url.py`. The client that will later hit the network should still be testable as string in, string out.

```
# ticket_url.py
from urllib.parse import urlencode, urljoin
from urllib.request import Request


def ticket_request(base, ticket_id):
    url = urljoin(base, "ticket")
    query = urlencode({"id": ticket_id})
    return Request(f"{url}?{query}", method="GET")


def main():
    req = ticket_request("http://desk.local/", "T-11")
    print(req.full_url)
```

```
    print(req.get_method())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_url.py
```

Output:

```
http://desk.local/ticket?id=T-11
GET
```

```
# test_ticket_url.py
from ticket_url import ticket_request


def test_ticket_request_query():
    req = ticket_request("http://desk.local/", "T-11")
    assert req.full_url == "http://desk.local/ticket?id=T-11"
    assert req.get_method() == "GET"
```

Run:

```
uv run --with pytest pytest test_ticket_url.py -q
```

Output (duration varies):

```
.                                                                        [100%]
1 passed in 0.01s
```

This is the test you write first. The socket test is next, not instead.

Case 2: A local server that shuts down

Save as `desk_http.py`. Port `0` means "pick a free one." The thread runs `serve_forever`. `urlopen` hits it. `shutdown` unblocks the thread. `join` waits. The process exits.

```
# desk_http.py
from http.server import BaseHTTPRequestHandler, HTTPServer
from threading import Thread
from urllib.request import urlopen
```

```
class TicketHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        body = b'{"id": "T-11", "table": 4}'
        self.send_response(200)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)

    def log_message(self, format, *args):
        return


def serve():
    server = HTTPServer(("127.0.0.1", 0), TicketHandler)
    thread = Thread(target=server.serve_forever)
    thread.start()
    return server, thread


def fetch_ticket(url):
    with urlopen(url, timeout=2) as resp:
        return resp.status, resp.read().decode()


def main():
    server, thread = serve()
    host, port = server.server_address
    try:
        status, body = fetch_ticket(f"http://{host}:{port}/ticket")
        print(status)
        print(body)
    finally:
        server.shutdown()
        thread.join(timeout=2)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_http.py
```

Output:

```
200
```

```
{"id": "T-11", "table": 4}
```

```
# test_desk_http.py
from desk_http import fetch_ticket, serve


def test_ticket_endpoint():
    server, thread = serve()
    host, port = server.server_address
    try:
        status, body = fetch_ticket(f"http://{host}:{port}/ticket")
        assert status == 200
        assert "T-11" in body
    finally:
        server.shutdown()
        thread.join(timeout=2)
```

Run:

```
uv run --with pytest pytest test_desk_http.py -q
```

Output (duration varies):

```
.                                                                        [100%]
1 passed in 0.05s
```

`finally` runs on failure too. Without it, a failed assert leaves a thread and a socket behind.

Case 3: A temp-dir ticket store

Save as `ticket_store.py`. JSON files under a directory. The integration is "filesystem plus encode."

```
# ticket_store.py
import json
from pathlib import Path


def save_ticket(root, ticket):
    path = Path(root) / f"{ticket['id']}.json"
    path.write_text(json.dumps(ticket) + "\n")
    return path


def load_ticket(root, ticket_id):
    path = Path(root) / f"{ticket_id}.json"
```

```
    return json.loads(path.read_text())


def main():
    import tempfile

    with tempfile.TemporaryDirectory() as root:
        save_ticket(root, {"id": "T-11", "table": 4})
        print(load_ticket(root, "T-11"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_store.py
```

Output:

```
{'id': 'T-11', 'table': 4}
```

```
# test_ticket_store.py
from ticket_store import load_ticket, save_ticket


def test_round_trip(tmp_path):
    save_ticket(tmp_path, {"id": "T-11", "table": 4})
    ticket = load_ticket(tmp_path, "T-11")
    assert ticket["table"] == 4
    assert (tmp_path / "T-11.json").is_file()
```

Run:

```
uv run --with pytest pytest test_ticket_store.py -q
```

Output (duration varies):

```
.                                                                        [100%]
1 passed in 0.01s
```

`tmp_path` is already unique per test. You do not need a second `TemporaryDirectory` inside the test.

The trap

This fragment never returns. Do not run it. `serve_forever` blocks the main thread. There is no `shutdown`, no `urlopen`, no `join`.

```
# do_not_run_forever.py
from http.server import BaseHTTPRequestHandler, HTTPServer


class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.end_headers()


def main():
    HTTPServer(("127.0.0.1", 8000), Handler).serve_forever()


if __name__ == "__main__":
    main()
```

The fix is Case 2: a thread, a request, `shutdown`, `join`. Bind `127.0.0.1` and port `0`. Leave port 8000 for humans.

The boring rule

- Prefer `urllib` (parse, request objects, `urlopen`) against a function until a socket is the point.
- Bind `127.0.0.1` and port `0`. Set a timeout on `urlopen`.
- `shutdown` and `join` in `finally`. The process must exit.
- `tmp_path` / `TemporaryDirectory` for desk files. Do not use the repo as a store.
- One real seam per test is enough. You do not need a Docker network to check JSON on disk.

Try this

1. Make `TicketHandler` return 404 unless the path is `/ticket`. Assert both status codes.
2. Add `load_ticket` raising `FileNotFoundError` when the JSON is missing. Test it with `tmp_path`.
3. Point `ticket_request` at the live server from Case 2 (`urlopen(req)`). Keep the shutdown in `finally`.
4. Change `save_ticket` to reject an id that contains `/`. Test that it does not write outside `root`.

Concurrency

Concurrency Basics

Concurrency Basics

Concurrency is overlapping waits. **Parallelism** is overlapping work. The boring default is a sequential program until a wait is real and measured. Threads help when the desk is idle on I/O (a request, a disk, a sleep). Processes help when the desk is busy on CPU (a long loop in Python). Do not start there.

Mental model

CPython's default build has a **global interpreter lock** (GIL): one thread at a time runs Python bytecode. Python 3.14 also ships a **free-threading** build that can run threads in parallel, but the interpreter `uv` gives you unless you ask otherwise is still the GIL build. So the boring rule does not change: threads overlap I/O; processes (or a free-threaded interpreter you chose on purpose) overlap CPU.

A sequential `for` over tickets is easier to test, easier to abort, and often fast enough. The next chapters add threads, `asyncio`, and processes. This one stays on one stack.

Worked examples

Case 1: One ticket at a time

Save as `sequential_rail.py`. The rail is a list. The function returns. No queue, no pool.

```
# sequential_rail.py
def fire(ticket):
    return f"fired {ticket}"


def main():
    tickets = ["T-11", "T-12", "T-13"]
    for ticket in tickets:
        print(fire(ticket))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python sequential_rail.py
```

Output:

```
fired T-11
fired T-12
fired T-13
```

This is the program you keep until `fire` actually waits on something outside the process.

Case 2: A wait you can see

Save as `sequential_wait.py`. `time.sleep` stands in for a slow printer. Three tickets, 0.05 seconds each, on one thread. The clock is the sum.

```
# sequential_wait.py
import time


def print_ticket(ticket):
    time.sleep(0.05)
    return f"printed {ticket}"


def main():
    tickets = ["T-11", "T-12", "T-13"]
    started = time.perf_counter()
    for ticket in tickets:
        print(print_ticket(ticket))
    elapsed = time.perf_counter() - started
    print(f"elapsed >= 0.15: {elapsed >= 0.15}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python sequential_wait.py
```

Output:

```
printed T-11
printed T-12
printed T-13
elapsed >= 0.15: True
```

The next chapter overlaps those sleeps with threads. The number to remember is: sequential I/O **adds**. Overlapped I/O **does not**, up to the number of waiters.

Case 3: CPU work is different

Save as `sequential_cpu.py`. Counting pence on a long list is CPU. Threads will not magically divide this on the default interpreter. Measure it as itself, then decide.

```
# sequential_cpu.py
def total_pence(orders):
    total = 0
    for pence in orders:
        total += pence
    return total


def main():
    tables = [
        [1200, 800, 400],
        [2000],
        [500, 500, 500],
    ]
    print(sum(total_pence(orders) for orders in tables))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python sequential_cpu.py
```

Output:

```
5900
```

A real hot loop might belong in `ProcessPoolExecutor` (later) or in a better algorithm. It does not belong in a thread pool "because we have cores" until you have a measurement.

The trap

Save as `thread_for_style.py`. Four threads, no I/O, no join discipline you can explain, and a result you still had to collect yourself. This is slower to read than Case 1 and no faster on CPU under the GIL.

```python
# thread_for_style.py
from threading import Thread


def fire(ticket, bucket):
    bucket.append(f"fired {ticket}")


def main():
    tickets = ["T-11", "T-12", "T-13"]
    bucket = []
    workers = [
        Thread(target=fire, args=(ticket, bucket)) for ticket in tickets
    ]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    for line in bucket:
        print(line)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python thread_for_style.py
```

Output (order of lines may vary):

```
fired T-11
fired T-12
fired T-13
```

You paid for scheduling and a shared list to reprint Case 1. If the order matters, you now have a bug. Stay sequential until a wait or a profile says otherwise.

The boring rule

- Write the sequential version first. Keep it if it is fast enough.
- Threads overlap **waiting**. Processes overlap **CPU** on the default CPython build.
- The GIL is still the boring default in 3.14. Free-threading is an opt-in build, not a reason to share mutable state.
- `join` every thread you start. A daemon thread that outlives `main` is how shutdown hangs.
- Shared lists and counters are the next chapter's trap, not a design.

Try this

1. In `sequential_wait.py`, drop `sleep` to `0.0` and see `elapsed >= 0.15` become `False`.
2. Add a fourth ticket. The sequential wait should stay above `0.20` seconds.
3. Change `sequential_cpu.py` to print each table total, then the grand total.
4. In `thread_for_style.py`, print `ticket` inside `fire` before append. Run it twice. Note the order.

Threading and Futures

Threading and Futures

A **thread** is another stack in the same process. `Thread(...).start()` runs a function. `.join()` waits until it finishes. A `Lock` makes a critical section exclusive. `ThreadPoolExecutor` is a bounded pile of workers plus a `Future` for each job. The boring default is the executor with a `with` block (it **shutdowns** and waits) and no shared mutable state if you can return a value instead.

Mental model

Use threads when `fire` spends time in I/O. Each worker still needs a join, or the executor's shutdown, or you leak work past `main`.

`Lock` around the smallest piece of shared data. If two threads append to a list or bump a counter, you need a lock or you need to stop sharing.

`concurrent.futures.ThreadPoolExecutor` maps a function over a collection and keeps the results in order when you use `.map`. Prefer it over a handwritten list of `Thread` objects.

Worked examples

Case 1: `Thread` and `join`

Save as `print_threads.py`. Three printers, each waits 0.05 seconds. Start all, then join all. The wall clock should beat the sequential 0.15 seconds from the previous chapter.

```
# print_threads.py
import time
from threading import Thread


def print_ticket(ticket, bucket):
    time.sleep(0.05)
    bucket.append(f"printed {ticket}")


def main():
    tickets = ["T-11", "T-12", "T-13"]
    bucket = []
    workers = [
        Thread(target=print_ticket, args=(ticket, bucket))
```

```
            for ticket in tickets
        ]
        started = time.perf_counter()
        for worker in workers:
            worker.start()
        for worker in workers:
            worker.join()
        elapsed = time.perf_counter() - started
        for line in bucket:
            print(line)
        print(f"elapsed < 0.15: {elapsed < 0.15}")


    if __name__ == "__main__":
        main()
```

Run:

```
uv run python print_threads.py
```

Output (order of `printed` lines may vary):

```
printed T-11
printed T-12
printed T-13
elapsed < 0.15: True
```

If you omit the `join` loop, `main` can print the elapsed time before the workers finish. Always join.

Case 2: `Lock` around a counter

Save as `cover_count.py`. Four workers each seat 1000 covers. The lock makes `+=` exclusive.

```
# cover_count.py
from threading import Lock, Thread


def seat(covers, lock, n):
    for _ in range(n):
        with lock:
            covers[0] += 1


def main():
    covers = [0]
    lock = Lock()
```

```
    workers = [
        Thread(target=seat, args=(covers, lock, 1000)) for _ in range(4)
    ]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    print(covers[0])


if __name__ == "__main__":
    main()
```

Run:

```
uv run python cover_count.py
```

Output:

```
4000
```

`covers` is a one-element list so the workers share the object. A lock around the increment is the whole design. Better: each worker returns `n` and the parent sums. That version needs no lock.

Case 3: `ThreadPoolExecutor`

Save as `print_pool.py`. The context manager calls `shutdown(wait=True)` on the way out. `.map` yields results in the **input** order, not the finish order.

```
# print_pool.py
import time
from concurrent.futures import ThreadPoolExecutor


def print_ticket(ticket):
    time.sleep(0.05)
    return f"printed {ticket}"


def main():
    tickets = ["T-11", "T-12", "T-13"]
    started = time.perf_counter()
    with ThreadPoolExecutor(max_workers=3) as pool:
        lines = list(pool.map(print_ticket, tickets))
    elapsed = time.perf_counter() - started
    for line in lines:
```

```
        print(line)
    print(f"elapsed < 0.15: {elapsed < 0.15}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python print_pool.py
```

Output:

```
printed T-11
printed T-12
printed T-13
elapsed < 0.15: True
```

`max_workers=3` bounds the pile. A pool of 3 for 3 tickets is fine. A pool of 500 for 3 tickets is not.

`pool.submit` returns a `Future`. `future.result()` waits and re-raises. `.map` is the boring form when you have one function and a list.

The trap

Save as `cover_race.py`. Same idea as Case 2, no lock. Each worker reads the counter, yields (`time.sleep(0)`), then writes. Four threads, 2000 seats each. The expected total is 8000. You will not get it.

```
# cover_race.py
import time
from threading import Thread


def seat(covers, n):
    for _ in range(n):
        current = covers[0]
        time.sleep(0)
        covers[0] = current + 1


def main():
    covers = [0]
    workers = [
        Thread(target=seat, args=(covers, 2000)) for _ in range(4)
    ]
```

```
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    print(covers[0])


if __name__ == "__main__":
    main()
```

Run:

```
uv run python cover_race.py
```

Output (yours will differ; **not** 8000):

```
2001
```

The number is a symptom. The bug is shared mutable state without a lock. `time.sleep(0)` only makes the lost update easy to see. Put the lock back, or do not share: return `n` from each worker.

The boring rule

- `start` then `join`. The executor `with` block is join-plus-shutdown.
- Bound `max_workers`. Do not `Thread()` in an unbounded loop.
- Lock only the shared counter or list. Prefer returning a value.
- `.map` when order of results should match the inputs.
- Threads for I/O waits. Do not start a thread per ticket because it looks busy.

Try this

1. In `print_threads.py`, comment out the `join` loop. Run it. Notice missing lines or a tiny elapsed time.
2. Change Case 2 so `seat` returns `n` and the parent sums. Delete the lock.
3. Use `pool.submit` and `future.result()` instead of `.map` for one ticket. Print that one line.
4. Set `max_workers=1` in `print_pool.py`. `elapsed < 0.15` should become `False`.

asyncio

asyncio

`async def` marks a **coroutine function**. Calling it does not run the body; it builds a coroutine object. `await` pauses this coroutine until that one finishes. `asyncio.run` is the boring entry point: it starts the event loop, runs one coroutine to completion, and closes the loop. Use asyncio when you have **many I/O waits** in one process and you want to overlap them without a thread per wait.

Mental model

The loop runs one coroutine until it `await`s. Then it can run another. `asyncio.sleep` is a fake I/O wait. Real I/O uses libraries that actually yield (sockets, subprocess, some HTTP clients). `time.sleep` inside a coroutine blocks the **loop**. Do not do that.

`asyncio.TaskGroup` starts child tasks and waits for all of them. If one fails, the group cancels the others and raises an `ExceptionGroup`.

`asyncio.timeout` cancels the block when the budget is gone. It raises `TimeoutError`. There is no infinite `while True` in this chapter.

Worked examples

Case 1: `async def`, `await`, `asyncio.run`

Save as `async_fire.py`. `await asyncio.sleep` yields the loop. Sequential `await`s still add, because you waited for each ticket before starting the next.

```
# async_fire.py
import asyncio


async def fire(ticket):
    await asyncio.sleep(0.05)
    return f"fired {ticket}"


async def main():
    for ticket in ["T-11", "T-12"]:
        print(await fire(ticket))
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Run:

```
uv run python async_fire.py
```

Output:

```
fired T-11
fired T-12
```

`asyncio.run(main())` belongs under the `main` guard, like any other script entry. Do not call `asyncio.run` twice in one process if you can avoid it.

Case 2: `TaskGroup` overlaps waits

Save as `async_shifts.py`. Both shifts sleep at the same time. The group does not exit until both tasks finish. `.result()` is safe after the `async with` block.

```
# async_shifts.py
import asyncio
import time


async def load_shift(name, delay):
    await asyncio.sleep(delay)
    return f"{name} ready"


async def main():
    started = time.perf_counter()
    async with asyncio.TaskGroup() as tg:
        am = tg.create_task(load_shift("am", 0.05))
        pm = tg.create_task(load_shift("pm", 0.05))
    elapsed = time.perf_counter() - started
    print(am.result())
    print(pm.result())
    print(f"elapsed < 0.10: {elapsed < 0.10}")


if __name__ == "__main__":
    asyncio.run(main())
```

Run:

```
uv run python async_shifts.py
```

Output:

```
am ready
pm ready
elapsed < 0.10: True
```

`asyncio.gather` is the older pile of awaitables. `TaskGroup` is the 3.11+ default: structured, cancels siblings on failure.

Case 3: `timeout`

Save as `async_timeout.py`. The grill takes too long. The timeout cancels the wait. The program still exits.

```
# async_timeout.py
import asyncio


async def grill(ticket):
    await asyncio.sleep(1)
    return f"done {ticket}"


async def main():
    try:
        async with asyncio.timeout(0.05):
            print(await grill("T-11"))
    except TimeoutError:
        print("grill too slow")


if __name__ == "__main__":
    asyncio.run(main())
```

Run:

```
uv run python async_timeout.py
```

Output:

```
grill too slow
```

`TimeoutError` here is the built-in exception. Bound every external wait that can hang. A kitchen that never answers should not pin your process.

Case 4: Many tickets, one group

Save as `async_rail.py`. Create a task per ticket inside the group. Print in a stable order after the group completes.

```
# async_rail.py
import asyncio


async def fire(ticket):
    await asyncio.sleep(0.01)
    return f"fired {ticket}"


async def main():
    tickets = ["T-11", "T-12", "T-13"]
    async with asyncio.TaskGroup() as tg:
        tasks = [tg.create_task(fire(t)) for t in tickets]
    for task in tasks:
        print(task.result())


if __name__ == "__main__":
    asyncio.run(main())
```

Run:

```
uv run python async_rail.py
```

Output:

```
fired T-11
fired T-12
fired T-13
```

The tasks may finish in any order. You print from the list you created, so the report is stable.

The trap

Save as `async_sleep_wrong.py`. `time.sleep` blocks the thread the loop runs on. The second ticket cannot start until the first sleep is over. You wrote `async` and got sequential blocking I/O with extra syntax.

```
# async_sleep_wrong.py
import asyncio
import time
```

```
async def fire(ticket):
    time.sleep(0.05)
    return f"fired {ticket}"


async def main():
    started = time.perf_counter()
    async with asyncio.TaskGroup() as tg:
        tg.create_task(fire("T-11"))
        tg.create_task(fire("T-12"))
    elapsed = time.perf_counter() - started
    print(f"elapsed >= 0.10: {elapsed >= 0.10}")


if __name__ == "__main__":
    asyncio.run(main())
```

Run:

```
uv run python async_sleep_wrong.py
```

Output:

```
elapsed >= 0.10: True
```

Replace `time.sleep` with `await asyncio.sleep` and the elapsed check flips. `async def` is not a speed potion. Only `await` points overlap.

The boring rule

- `asyncio.run(main())` once, under the `main` guard.
- `await` real I/O (or `asyncio.sleep` in examples). Never `time.sleep` in a coroutine.
- `TaskGroup` for a known set of child tasks. Do not fire-and-forget.
- `asyncio.timeout` around waits that can hang. The process must still exit.
- Sequential `await` in a `for` is still sequential. Overlap needs tasks.

Try this

1. In `async_fire.py`, create both `fire` tasks in a `TaskGroup` and print `.result()` after the block.
2. Change Case 3's timeout to `2` seconds so the grill succeeds and prints `done T-11`.
3. Raise `RuntimeError("burnt")` in one `load_shift`. Catch `ExceptionGroup` around the `TaskGroup`.

4. Add a third shift `late` to Case 2. Keep the elapsed check under `0.10`.

multiprocessing

multiprocessing

A **process** is a separate interpreter with its own memory. `ProcessPoolExecutor` runs a function in those interpreters and brings the return value back. Use it for **CPU** work the GIL would serialise in threads. The boring default is a module-level worker, a `with` pool, and a mandatory `if __name__ == "__main__":` guard.

Mental model

On Python 3.14 the default start method on Linux is **forkserver** (not `fork`). Child processes import your module and look up the worker by name. The worker must be a top-level function. A lambda or a nested `def` will not unpickle.

The `main` guard stops the child import from starting **another** pool. Without it, you can spawn processes until the machine weeps. Always write the guard. Always put pool construction inside `main`.

Arguments and return values must pickle. Lists of ints are fine. Open sockets and generators are not.

Worked examples

Case 1: `ProcessPoolExecutor` and the guard

Save as `table_totals.py`. Each table's orders are summed in a child. `pool.map` preserves input order. The `with` block shuts the pool down.

```
# table_totals.py
from concurrent.futures import ProcessPoolExecutor


def total_for_table(orders):
    return sum(orders)


def main():
    tables = [
        [1200, 800, 400],
        [2000],
        [500, 500, 500],
```

```
    ]
    with ProcessPoolExecutor(max_workers=3) as pool:
        totals = list(pool.map(total_for_table, tables))
    print(totals)
    print(sum(totals))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python table_totals.py
```

Output:

```
[2400, 2000, 1500]
5900
```

`total_for_table` sits at module level on purpose. The children import `table_totals` and call that name.

Case 2: CPU work that is actually worth a process

Save as `cover_hot.py`. A tight Python loop is GIL-bound in threads. Processes run it on more than one core. The result is still a number you check.

```
# cover_hot.py
from concurrent.futures import ProcessPoolExecutor


def count_covers(n):
    total = 0
    for i in range(n):
        total += i % 7
    return total


def main():
    jobs = [200_000, 200_000, 200_000]
    with ProcessPoolExecutor(max_workers=3) as pool:
        parts = list(pool.map(count_covers, jobs))
    print(parts)
    print(sum(parts))
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python cover_hot.py
```

Output:

```
[599994, 599994, 599994]
1799982
```

Three tiny jobs like this may lose to sequential overhead. That is fine: the pattern is what you copy when a profiler says the loop is hot. Bound `max_workers` to something near the core count, not to the number of tickets in a year.

Case 3: What you send across the boundary

Save as `ticket_payload.py`. Send plain data. Get plain data back. Do not send a live `TicketWindow` object unless you are ready to pickle its guts.

```
# ticket_payload.py
from concurrent.futures import ProcessPoolExecutor


def label(ticket):
    return f"{ticket['id']} table {ticket['table']}"


def main():
    tickets = [
        {"id": "T-11", "table": 4},
        {"id": "T-12", "table": 2},
    ]
    with ProcessPoolExecutor(max_workers=2) as pool:
        for line in pool.map(label, tickets):
            print(line)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_payload.py
```

Output:

```
T-11 table 4
T-12 table 2
```

A dict of strings and ints pickles. A generator of tickets does not travel as a live cursor; you would send a list (or a path the child reopens).

The trap

Save as `nested_worker.py`. The worker is nested inside `main`. forkserver cannot look it up on the child.

```python
# nested_worker.py
from concurrent.futures import ProcessPoolExecutor


def main():
    def total_for_table(orders):
        return sum(orders)

    tables = [[1200, 800], [2000]]
    with ProcessPoolExecutor(max_workers=2) as pool:
        print(list(pool.map(total_for_table, tables)))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python nested_worker.py
```

Output (traceback shortened; the process exits non-zero):

```
_pickle.PicklingError: Can't pickle local object <function main.<locals>.total_for_table at 0x.
```

The child cannot look up a nested function. Lift `total_for_table` to module level. Keep the guard. Do not wrap the whole file in `main` without leaving the worker outside.

A second trap: calling `ProcessPoolExecutor` at import time, above the guard. Children import the module, start more children, and you have a fork bomb. Construction stays inside `main`.

The boring rule

- `if __name__ == "__main__":` on every process-pool script. No exceptions.
- Worker functions at module level. No nested `def`, no lambda.
- Send lists, dicts, numbers, strings. Not sockets, not generators, not the executor itself.
- `with ProcessPoolExecutor(max_workers=...)` so shutdown always runs.
- Threads for I/O, processes for CPU. Do not open a process pool to print three tickets.

Try this

1. Move `total_for_table` in Case 1 to nested-inside-`main` and confirm it breaks. Move it back.
2. Add a fourth table to `table_totals.py`. Keep `max_workers=3`.
3. Change `label` to reject a missing `table` key with `ValueError`. See how the parent surfaces that error from `.map`.
4. Run `cover_hot.py` with `max_workers=1` and with `max_workers=3`. Feel the wall-clock difference on your machine; do not expect a textbook speedup on tiny jobs.

Concurrency Design Guidelines

Concurrency Design Guidelines

Concurrency is a tool you add after a sequential program is correct. The boring defaults: **do not share mutable state**, **bound the work**, and **stay sequential until you have a measurement**. This chapter shows a leak, then the fix, three times.

Mental model

Shared memory is the bug factory. If two threads own a list, you now own a lock story. If a parent can get a return value instead, do that.

Unbounded `Thread()` or `create_task` in a `for` over an open-ended rail will take the process down. A pool with `max_workers` (or a `TaskGroup` over a slice) is a budget.

`time.perf_counter` around the sequential version is cheaper than a redesign. If 3 milliseconds became 4 with threads, you bought noise.

Worked examples

Case 1: Leak — shared list; fix — return a value

Save as `shared_rail.py`. Workers append to one list with no lock. The length is often right; the contents and order are not a contract.

```
# shared_rail.py
from threading import Thread


def fire(ticket, bucket):
    bucket.append(f"fired {ticket}")


def main():
    tickets = ["T-11", "T-12", "T-13"]
    bucket = []
    workers = [Thread(target=fire, args=(t, bucket)) for t in tickets]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
```

```
    print(bucket)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shared_rail.py
```

Output (order may vary):

```
['fired T-11', 'fired T-12', 'fired T-13']
```

Save as `return_rail.py`. The executor owns the workers. Each call returns a string. The parent prints in input order. No shared list.

```
# return_rail.py
from concurrent.futures import ThreadPoolExecutor


def fire(ticket):
    return f"fired {ticket}"


def main():
    tickets = ["T-11", "T-12", "T-13"]
    with ThreadPoolExecutor(max_workers=3) as pool:
        for line in pool.map(fire, tickets):
            print(line)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python return_rail.py
```

Output:

```
fired T-11
fired T-12
fired T-13
```

If you need a running total, sum the returned numbers in the parent. Do not publish a counter every thread bumps.

Case 2: Leak — unbounded threads; fix — a bound pool

Save as `unbounded.py`. One thread per ticket, no cap. Fine for three. Fatal for a file with 50_000 lines.

```
# unbounded.py
from threading import Thread


def fire(ticket, bucket):
    bucket.append(ticket)


def main():
    tickets = [f"T-{n}" for n in range(3)]
    bucket = []
    workers = []
    for ticket in tickets:
        worker = Thread(target=fire, args=(ticket, bucket))
        worker.start()
        workers.append(worker)
    for worker in workers:
        worker.join()
    print(len(bucket))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python unbounded.py
```

Output:

```
3
```

Save as `bounded.py`. `max_workers=2` is the budget. The rest of the rail waits in the queue inside the executor.

```
# bounded.py
from concurrent.futures import ThreadPoolExecutor


def fire(ticket):
    return ticket
```

```
def main():
    tickets = [f"T-{n}" for n in range(3)]
    with ThreadPoolExecutor(max_workers=2) as pool:
        print(len(list(pool.map(fire, tickets))))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python bounded.py
```

Output:

```
3
```

Same three tickets. The bound is the design. Pick `max_workers` from the I/O you are waiting on (a handful), not from the size of the input.

Case 3: Leak — threads for a cheap loop; fix — sequential until measured

Save as `needless_pool.py`. The work is a few additions. The pool is theatre.

```
# needless_pool.py
from concurrent.futures import ThreadPoolExecutor


def total_for_table(orders):
    return sum(orders)


def main():
    tables = [[1200, 800], [400], [500, 500]]
    with ThreadPoolExecutor(max_workers=3) as pool:
        print(sum(pool.map(total_for_table, tables)))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python needless_pool.py
```

Output:

```
3400
```

Save as `measured.py`. Same arithmetic. One stack. If this is slow, the next move is a better loop or a process pool after a timer, not more threads.

```python
# measured.py
import time


def total_for_table(orders):
    return sum(orders)


def main():
    tables = [[1200, 800], [400], [500, 500]]
    started = time.perf_counter()
    grand = sum(total_for_table(orders) for orders in tables)
    elapsed = time.perf_counter() - started
    print(grand)
    print(f"elapsed < 0.01: {elapsed < 0.01}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python measured.py
```

Output:

```
3400
elapsed < 0.01: True
```

When `elapsed < 0.01` is true, you do not have a concurrency problem.

The trap

Mixing all three leaks: a shared dict, a new `Thread` per key, no join on error paths, no bound. The fix is not a bigger lock. The fix is Case 1's return values, Case 2's pool, and Case 3's timer, in that order.

Save as `three_leaks.py` only as a warning. Prefer not to run it as a template.

```python
# three_leaks.py
from threading import Thread
```

```
def bump(counts, table):
    counts[table] = counts.get(table, 0) + 1


def main():
    counts = {}
    workers = []
    for table in [4, 2, 4, 2, 4]:
        worker = Thread(target=bump, args=(counts, table))
        worker.start()
        workers.append(worker)
    for worker in workers:
        worker.join()
    print(counts)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python three_leaks.py
```

Output (you might get this; you might not):

```
{4: 3, 2: 2}
```

`dict.get` plus assign is not atomic. The print can lie. The sequential fix:

```
# three_fixes.py
from collections import Counter


def main():
    tables = [4, 2, 4, 2, 4]
    print(dict(Counter(tables)))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python three_fixes.py
```

Output:

```
{4: 3, 2: 2}
```

No threads. Same answer every run. That is the design.

The boring rule

- Do not share mutable state. Return values; sum in the parent.
- Bound workers (`max_workers`, a `TaskGroup` over a known list). Join or shutdown on every path.
- Sequential until `perf_counter` says you are waiting or burning CPU.
- Threads for I/O, processes for CPU, asyncio for many I/O waits in one process. Pick one style per program.
- A green print from a racy program is not a test. If order or totals matter, make them deterministic.

Try this

1. Add a `Lock` to `shared_rail.py`. Confirm it still does not give you input order. Then switch to `return_rail.py`.
2. Change `bounded.py` to 20 tickets and `max_workers=4`. Print each line from `.map`.
3. Time `needless_pool.py` and `measured.py` with `perf_counter`. Keep the faster one.
4. Rewrite `three_leaks.py` with `ThreadPoolExecutor` and a worker that returns `table`. Count in the parent with `Counter`.

Standard Library

pathlib and Files

pathlib and Files

The boring way to touch disk in Python is **pathlib.Path**: a path is an object, / joins parts, and read_text / write_text carry an encoding. Run file examples inside **tempfile.TemporaryDirectory** so they create nothing you have to clean up.

Mental model

A **path** is a location (tickets/t7.txt), not the file's bytes. Path stores that location. Joining is /, not string concatenation. Reading and writing text is read_text and write_text with encoding="utf-8" spelled out — never "whatever the locale is today."

TemporaryDirectory makes a real directory, yields its path as a string, and deletes the tree when the with block ends. That is how every listing in this chapter stays runnable and clean.

Path is the default. Reach for os.path only when an old API demands a string and will not take a Path.

Worked examples

Case 1: Write a ticket, read it back

Save as save_ticket.py. write_text creates the file. read_text returns a str.

```
# save_ticket.py
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        desk = Path(raw)
        ticket = desk / "ticket-7.txt"
        ticket.write_text("ticket 7 → table 12\n", encoding="utf-8")
        print(ticket.name)
        print(ticket.read_text(encoding="utf-8"), end="")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python save_ticket.py
```

Output:

```
ticket-7.txt
ticket 7 → table 12
```

The temp directory is gone after `main` returns. The program still proved the round-trip.

Case 2: Directories, `iterdir`, and `glob`

Save as `desk_tree.py`. `mkdir` creates a folder. `iterdir` lists it. `glob` matches a pattern.

```
# desk_tree.py
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        desk = Path(raw)
        tickets = desk / "tickets"
        tickets.mkdir()
        (tickets / "t1.txt").write_text("table 3\n", encoding="utf-8")
        (tickets / "t2.txt").write_text("table 11\n", encoding="utf-8")
        names = sorted(p.name for p in tickets.iterdir())
        print(names)
        for path in sorted(desk.glob("tickets/*.txt")):
            print(f"{path.name}: {path.read_text(encoding='utf-8').strip()}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_tree.py
```

Output:

```
['t1.txt', 't2.txt']
t1.txt: table 3
t2.txt: table 11
```

Sort names if you care about stable output. Directory order is not a promise.

Case 3: Missing files fail out loud

Save as `missing_ticket.py`. `read_text` raises `FileNotFoundError`. Catch that exception. Do not return `None` as a secret signal.

```
# missing_ticket.py
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        path = Path(raw) / "missing.txt"
        try:
            path.read_text(encoding="utf-8")
        except FileNotFoundError:
            print("no ticket file")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python missing_ticket.py
```

Output:

```
no ticket file
```

Case 4: Nested paths need parents

Save as `nested_shift.py`. `write_text` does not create missing parent directories. `mkdir(parents=True, exist_ok=True)` does.

```
# nested_shift.py
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        path = Path(raw) / "shifts" / "monday" / "note.txt"
        path.parent.mkdir(parents=True, exist_ok=True)
        path.write_text("front desk\n", encoding="utf-8")
        print(path.parent.name)
        print(path.read_text(encoding="utf-8"), end="")
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python nested_shift.py
```

Output:

```
monday
front desk
```

The trap

`Path / other` looks like join. If `other` is **absolute**, the left side is discarded. This program does not write anywhere; it only prints what the join produced.

Save as `abs_join.py`:

```
# abs_join.py
from pathlib import Path


def main() -> None:
    desk = Path("/home/desk")
    wrong = desk / "/tickets/t7.txt"
    right = desk / "tickets" / "t7.txt"
    print(wrong)
    print(right)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python abs_join.py
```

Output:

```
/tickets/t7.txt
/home/desk/tickets/t7.txt
```

The first line is not under the desk. A leading `/` on the right-hand part is a new root, not a child. Join **relative** pieces: `"tickets"` then `"t7.txt"`.

The boring rule

- Use `Path`. Join with `/` and relative names.
- Always pass `encoding="utf-8"` to `read_text`, `write_text`, and `open`.
- Create parent directories explicitly (`mkdir(parents=True, exist_ok=True)`).
- Use `TemporaryDirectory` in examples and tests so leftover files are not the API.
- Catch `FileNotFoundError` / `PermissionError` at the edge, not as `except Exception`.

Try this

1. In `save_ticket.py`, write two tickets and print how many `.txt` files `desk.glob("*.txt")` finds.
2. In `desk_tree.py`, add `tickets / "archive"` with `mkdir` and skip directories when you print tables (`if path.is_file()`).
3. In `nested_shift.py`, try `write_text` **before** `mkdir` and catch `FileNotFoundError`.

I/O and Streaming

I/O and Streaming

The boring default for text is a **file object** you iterate: one line at a time, closed by a `with` block. `io.StringIO` is the same shape in memory, which is how you test a writer without touching disk.

Mental model

A **text stream** (`TextIO`) is something you can `read`, `write`, and iterate as lines. Real files from `Path.open` are streams. `io.StringIO` is a stream backed by a string. Functions that take `TextIO` work on both.

Streaming means you process a chunk (usually a line) and let it go. `read()` pulls the whole file into one `str`. That is fine for a 20-line ticket list. It is a habit that blows up on a month of logs.

`tempfile.TemporaryDirectory` still owns any real files in this chapter.

Worked examples

Case 1: `StringIO` as a fake report file

Save as `report_buffer.py`. The writer only knows it has a stream. `getvalue()` is the in-memory extra.

```
# report_buffer.py
from io import StringIO


def write_shift_report(buf: StringIO, tables: list[int]) -> None:
    buf.write("shift report\n")
    for n in tables:
        buf.write(f"table {n} open\n")


def main() -> None:
    buf = StringIO()
    write_shift_report(buf, [3, 11, 12])
    print(buf.getvalue(), end="")


if __name__ == "__main__":
```

```
    main()
```

Run:

```
uv run python report_buffer.py
```

Output:

```
shift report
table 3 open
table 11 open
table 12 open
```

Case 2: Iterate a real file by line

Save as `iter_tickets.py`. `for line in fh` reads incrementally. `split` is enough for this format.

```
# iter_tickets.py
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        path = Path(raw) / "tickets.txt"
        path.write_text("7 12\n8 3\n9 11\n", encoding="utf-8")
        with path.open(encoding="utf-8") as fh:
            for line in fh:
                ticket_id, table = line.split()
                print(f"ticket {ticket_id} → table {table}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python iter_tickets.py
```

Output:

```
ticket 7 → table 12
ticket 8 → table 3
ticket 9 → table 11
```

The `with` closes the file even if a later line raises.

Case 3: Accept any `TextIO`

Save as `copy_tables.py`. `typing.TextIO` is the type for a text stream. The function copies without caring whether the ends are files or `StringIO`.

```
# copy_tables.py
from io import StringIO
from typing import TextIO


def copy_open_tables(src: TextIO, dest: TextIO) -> int:
    n = 0
    for line in src:
        dest.write(line)
        n += 1
    return n


def main() -> None:
    src = StringIO("3\n11\n")
    dest = StringIO()
    count = copy_open_tables(src, dest)
    print(count)
    print(dest.getvalue(), end="")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python copy_tables.py
```

Output:

```
2
3
11
```

Case 4: Stream from a temp file into `StringIO`

Save as `file_to_buffer.py`. Same helper, real file on the left.

```
# file_to_buffer.py
from io import StringIO
from pathlib import Path
from tempfile import TemporaryDirectory
```

```
from typing import TextIO


def copy_open_tables(src: TextIO, dest: TextIO) -> int:
    n = 0
    for line in src:
        dest.write(line)
        n += 1
    return n


def main() -> None:
    with TemporaryDirectory() as raw:
        path = Path(raw) / "open.txt"
        path.write_text("12\n4\n", encoding="utf-8")
        dest = StringIO()
        with path.open(encoding="utf-8") as src:
            copy_open_tables(src, dest)
        print(dest.getvalue(), end="")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python file_to_buffer.py
```

Output:

```
12
4
```

The trap

`read()` then `splitlines()` works until the file is large. This program slurps on purpose. Prefer the `for line in fh` loop from Case 2 when the file can grow.

Save as `slurp_tickets.py`:

```
# slurp_tickets.py
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
```

```
    with TemporaryDirectory() as raw:
        path = Path(raw) / "tickets.txt"
        path.write_text("7 12\n8 3\n", encoding="utf-8")
        text = path.read_text(encoding="utf-8")
        for line in text.splitlines():
            ticket_id, table = line.split()
            print(f"ticket {ticket_id} → table {table}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python slurp_tickets.py
```

Output:

```
ticket 7 → table 12
ticket 8 → table 3
```

The output matches the streaming version. The cost does not. Keep `read_text` for small configs. Iterate logs and exports.

The boring rule

- Type writers as `TextIO` (or `StringIO` when you need `getvalue`).
- Iterate lines for anything that might grow.
- Always `open(..., encoding="utf-8")`.
- Close with `with`. Do not rely on the garbage collector.
- Use `StringIO` in tests; use `TemporaryDirectory` when you need a real path.

Try this

1. In `report_buffer.py`, add a final line `tables: N` where `N` is `len(tables)`.
2. In `iter_tickets.py`, skip blank lines (`if not line.strip(): continue`).
3. Change `copy_open_tables` so it uppercases each line as it copies.

datetime and time

datetime and time

Store instants as **timezone-aware UTC**. Display them in a desk zone with `zoneinfo`. Do not treat naive "local time" as something you can save and compare later.

Mental model

`datetime.datetime` is a calendar stamp: year, month, day, hour, minute, second. **Naive** means `tzinfo` is `None` — a wall clock with no location. **Aware** means it knows its offset.

`datetime.timezone.utc` is the boring zone for storage and for `datetime.now`. `zoneinfo.ZoneInfo` names an IANA zone (`America/Chicago`) for display and for "what time is it at this desk."

`timedelta` is a duration (eight hours in a shift), not a clock.

The `time` module is clocks and sleeps (`time.monotonic`, `time.sleep`). It is not where you keep "Tuesday 14:30." Use `datetime` for that.

Worked examples

Case 1: An aware instant, UTC in, desk zone out

Save as `shift_open.py`. The stored value is UTC. The desk clock is a conversion.

```
# shift_open.py
from datetime import datetime, timezone
from zoneinfo import ZoneInfo


def main() -> None:
    opened = datetime(2026, 9, 7, 14, 30, tzinfo=timezone.utc)
    desk_tz = ZoneInfo("America/Chicago")
    print(opened.isoformat())
    print(opened.astimezone(desk_tz).isoformat())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_open.py
```

Output:

```
2026-09-07T14:30:00+00:00
2026-09-07T09:30:00-05:00
```

`isoformat()` is the string you put in JSON and logs.

Case 2: `timedelta` for a shift length

Save as `shift_length.py`. Add a duration to an aware datetime. The result stays aware.

```
# shift_length.py
from datetime import datetime, timedelta, timezone


def main() -> None:
    opened = datetime(2026, 9, 7, 14, 30, tzinfo=timezone.utc)
    closed = opened + timedelta(hours=8)
    print(closed.isoformat())
    print(str(closed - opened))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_length.py
```

Output:

```
2026-09-07T22:30:00+00:00
8:00:00
```

Case 3: Parse the ISO string you stored

Save as `parse_opened.py`. `fromisoformat` keeps the offset. `+00:00` comes back as `timezone.utc`.

```
# parse_opened.py
from datetime import datetime, timezone


def main() -> None:
    stored = "2026-09-07T14:30:00+00:00"
```

```
    opened = datetime.fromisoformat(stored)
    print(opened.tzinfo == timezone.utc)
    print(opened.hour, opened.minute)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python parse_opened.py
```

Output:

```
True
14 30
```

Case 4: "Now" is UTC when you ask for now

Save as `utc_now.py`. `datetime.now(timezone.utc)` is the call. Do not use `datetime.now()` with no arguments for something you will store.

```
# utc_now.py
from datetime import datetime, timezone


def main() -> None:
    now = datetime.now(timezone.utc)
    print(now.tzinfo == timezone.utc)
    print(now.utcoffset().total_seconds() == 0)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python utc_now.py
```

Output:

```
True
True
```

The clock value changes every run. The zone does not. Assert the zone, not the digits.

The trap

Naive and aware datetimes do not mix. Stamping a local wall time as UTC is a silent five-hour lie.

Save as `naive_trap.py`:

```
# naive_trap.py
from datetime import datetime, timezone
from zoneinfo import ZoneInfo


def main() -> None:
    naive = datetime(2026, 9, 7, 9, 30)
    aware = datetime(2026, 9, 7, 14, 30, tzinfo=timezone.utc)
    try:
        print(naive < aware)
    except TypeError as exc:
        print(type(exc).__name__)
    wrong = datetime(2026, 9, 7, 9, 30, tzinfo=timezone.utc)
    right = datetime(
        2026, 9, 7, 9, 30, tzinfo=ZoneInfo("America/Chicago")
    ).astimezone(timezone.utc)
    print(wrong.isoformat())
    print(right.isoformat())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python naive_trap.py
```

Output:

```
TypeError
2026-09-07T09:30:00+00:00
2026-09-07T14:30:00+00:00
```

`wrong` is "09:30 UTC." `right` is "09:30 at the Chicago desk," stored as UTC. Those are different instants. Attach the real zone first, then convert to UTC to store.

The boring rule

- Store UTC (`timezone.utc`). Display with `ZoneInfo`.

- Never save `datetime.now()` with no `tzinfo` as the system of record.
- Parse ISO-8601 with an offset (`fromisoformat`). Reject naive strings at the boundary.
- Use `timedelta` for durations. Do not add raw hours onto a naive local clock.
- `time.monotonic()` is for measuring elapsed seconds, not for timestamps.

Try this

1. In `shift_length.py`, print the close time in `America/Chicago`.
2. In `parse_opened.py`, parse `2026-09-07T09:30:00-05:00` and print UTC `isoformat()`.
3. Replace `timezone.utc` in `utc_now.py` with `ZoneInfo("UTC")` and print whether `tzinfo == timezone.utc` (it may not — compare `utcoffset()` instead).

JSON, TOML, and Encoding

JSON, TOML, and Encoding

The standard library already speaks the three formats a desk service actually ships: **JSON** for APIs and files, **TOML** for config you read, **CSV** for spreadsheets. Encode times as ISO strings. Do not invent a fourth format this week.

Mental model

JSON (`json`) is nested dicts, lists, strings, numbers, booleans, and `null`. It has no datetime type. You store `isoformat()` text.

TOML is the usual `pyproject.toml` / app-config language. `tomllib` **reads** it (`load` from bytes, `loads` from `str`). It does not write. Writing TOML is a third-party job, or you keep the file as a string in tests.

CSV is rows. Open CSV files with `newline=""` and an encoding. `csv.DictWriter` / `DictReader` keep headers honest.

Text files are UTF-8. `tomllib.load` wants a **binary** file (`"rb"`).

Worked examples

Case 1: JSON ticket round-trip

Save as `ticket_json.py`. `dumps` / `loads` for strings. Files use `write_text` with UTF-8.

```
# ticket_json.py
import json
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    ticket = {"id": 7, "table": 12, "status": "open"}
    with TemporaryDirectory() as raw:
        path = Path(raw) / "ticket.json"
        path.write_text(json.dumps(ticket, indent=2) + "\n", encoding="utf-8")
        loaded = json.loads(path.read_text(encoding="utf-8"))
        print(loaded["id"])
        print(loaded["status"])
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_json.py
```

Output:

```
7
open
```

Case 2: Read TOML config

Save as `desk_toml.py`. Write the file as bytes (or UTF-8 text), open `"rb"`, `tomllib.load`.

```
# desk_toml.py
import tomllib
from pathlib import Path
from tempfile import TemporaryDirectory

CONFIG = """
[desk]
name = "front"
tables = [3, 11, 12]
"""


def main() -> None:
    with TemporaryDirectory() as raw:
        path = Path(raw) / "desk.toml"
        path.write_bytes(CONFIG.encode("utf-8"))
        with path.open("rb") as fh:
            data = tomllib.load(fh)
        print(data["desk"]["name"])
        print(data["desk"]["tables"])


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_toml.py
```

Output:

```
front
[3, 11, 12]
```

`tomllib.loads(CONFIG)` is the same dict if you already have a string.

Case 3: CSV tickets

Save as `tickets_csv.py`. `newline=""` is required so the `csv` module owns line endings.

```python
# tickets_csv.py
import csv
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        path = Path(raw) / "tickets.csv"
        with path.open("w", encoding="utf-8", newline="") as fh:
            writer = csv.DictWriter(fh, fieldnames=["id", "table", "status"])
            writer.writeheader()
            writer.writerow({"id": 7, "table": 12, "status": "open"})
            writer.writerow({"id": 8, "table": 3, "status": "paid"})
        with path.open(encoding="utf-8", newline="") as fh:
            for row in csv.DictReader(fh):
                print(f"ticket {row['id']} table {row['table']} {row['status']}")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python tickets_csv.py
```

Output:

```
ticket 7 table 12 open
ticket 8 table 3 paid
```

Case 4: `tomllib` does not dump

Save as `toml_write.py`. The stdlib reader has no `dump`. Check, then keep writing JSON or a hand-made TOML string.

```
# toml_write.py
import tomllib


def main() -> None:
    print(hasattr(tomllib, "dump"))
    print(hasattr(tomllib, "load"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python toml_write.py
```

Output:

```
False
True
```

The trap

`json.dumps` will not serialize a `datetime`. Encode it yourself.

Save as `json_datetime.py`:

```
# json_datetime.py
import json
from datetime import datetime, timezone


def main() -> None:
    opened = datetime(2026, 9, 7, 14, 30, tzinfo=timezone.utc)
    try:
        json.dumps({"opened": opened})
    except TypeError as exc:
        print(type(exc).__name__)
        print(exc)
    print(json.dumps({"opened": opened.isoformat()}))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python json_datetime.py
```

Output:

```
TypeError
Object of type datetime is not JSON serializable
{"opened": "2026-09-07T14:30:00+00:00"}
```

The second line is the boring encoding. Parse it back with `datetime.fromisoformat`.

The boring rule

- JSON for data you exchange. TOML for human-edited config you read. CSV for tables other people open in a spreadsheet.
- UTF-8 everywhere. `tomllib.load` on `"rb"`. CSV with `newline=""`.
- Dates go to JSON as ISO-8601 strings with an offset.
- Do not add a TOML writer until you actually generate config files.
- `json.loads` / `tomllib.loads` raise on junk. Catch at the edge.

Try this

1. In `ticket_json.py`, add `"opened": "2026-09-07T14:30:00+00:00"` and print it after load.
2. In `desk_toml.py`, add `shift = "day"` under `[desk]` and print it.
3. In `tickets_csv.py`, add a third row and print `status` counts with a dict.

HTTP and Networking

HTTP and Networking

The boring HTTP client is **`urllib.request`**. The boring local server is **`http.server.HTTPServer`**. Start the server in a thread, make one request, **`shutdown`**, and exit. Do not leave `serve_forever` on the main thread. Do not call out to the public internet for a test.

Mental model

HTTP is request/response over a socket. `urlopen` is a GET (or a `Request` you configure). A successful body is bytes; decode it. A 404 is `urllib.error.HTTPError`, not an empty string.

`HTTPServer((host, port), Handler)` binds a socket. Port `0` means "pick a free port." `serve_forever` blocks, so it belongs on a **thread**. `server.shutdown()` from the main thread asks that loop to stop. `join` the thread. The process must exit.

`BaseHTTPRequestHandler` implements `do_GET` / `do_POST`. Override `log_message` if you do not want the default stderr access log.

Worked examples

Case 1: Local GET, then shutdown

Save as `desk_http.py`. The handler writes a small body. The client reads it. Then the server dies.

```
# desk_http.py
from http.server import BaseHTTPRequestHandler, HTTPServer
from threading import Thread
from urllib.request import urlopen


class DeskHandler(BaseHTTPRequestHandler):
    def do_GET(self) -> None:
        body = b"desk is open"
        self.send_response(200)
        self.send_header("Content-Type", "text/plain; charset=utf-8")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)
```

```
    def log_message(self, format: str, *args: object) -> None:
        return


def main() -> None:
    server = HTTPServer(("127.0.0.1", 0), DeskHandler)
    thread = Thread(target=server.serve_forever, daemon=True)
    thread.start()
    host, port = server.server_address
    with urlopen(f"http://{host}:{port}/") as resp:
        print(resp.read().decode("utf-8"))
    server.shutdown()
    thread.join(timeout=2)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_http.py
```

Output:

```
desk is open
```

The program returns. If it hangs, you skipped `shutdown`.

Case 2: POST a ticket as JSON

Save as `post_ticket.py`. `Request` sets method, body, and headers. The handler reads `Content-Length` bytes.

```
# post_ticket.py
import json
from http.server import BaseHTTPRequestHandler, HTTPServer
from threading import Thread
from urllib.request import Request, urlopen


class TicketHandler(BaseHTTPRequestHandler):
    last = ""

    def do_POST(self) -> None:
        length = int(self.headers.get("Content-Length", "0"))
        TicketHandler.last = self.rfile.read(length).decode("utf-8")
```

```
        body = b"ok"
        self.send_response(200)
        self.send_header("Content-Type", "text/plain; charset=utf-8")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)

    def log_message(self, format: str, *args: object) -> None:
        return


def main() -> None:
    server = HTTPServer(("127.0.0.1", 0), TicketHandler)
    thread = Thread(target=server.serve_forever, daemon=True)
    thread.start()
    host, port = server.server_address
    payload = json.dumps({"id": 7, "table": 12}).encode("utf-8")
    req = Request(
        f"http://{host}:{port}/tickets",
        data=payload,
        method="POST",
        headers={"Content-Type": "application/json"},
    )
    with urlopen(req) as resp:
        print(resp.read().decode("utf-8"))
    print(TicketHandler.last)
    server.shutdown()
    thread.join(timeout=2)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python post_ticket.py
```

Output:

```
ok
{"id": 7, "table": 12}
```

Case 3: 404 is an exception

Save as `http_missing.py`. `urlopen` raises `HTTPError` for 404. Read the body from the exception.

```
# http_missing.py
from http.server import BaseHTTPRequestHandler, HTTPServer
from threading import Thread
from urllib.error import HTTPError
from urllib.request import urlopen


class DeskHandler(BaseHTTPRequestHandler):
    def do_GET(self) -> None:
        body = b"missing"
        self.send_response(404)
        self.send_header("Content-Type", "text/plain; charset=utf-8")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)

    def log_message(self, format: str, *args: object) -> None:
        return


def main() -> None:
    server = HTTPServer(("127.0.0.1", 0), DeskHandler)
    thread = Thread(target=server.serve_forever, daemon=True)
    thread.start()
    host, port = server.server_address
    try:
        urlopen(f"http://{host}:{port}/nope")
    except HTTPError as exc:
        print(exc.code)
        print(exc.read().decode("utf-8"))
    server.shutdown()
    thread.join(timeout=2)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python http_missing.py
```

Output:

```
404
missing
```

The trap

`http.server.test()` and `serve_forever()` on the **main** thread never return. This program is the shape that exits. The broken shape is the same server without `shutdown` — do not ship that.

A second trap: `urlopen("https://httpbin.org/...")` in a unit test. That is a network dependency, a flaky clock, and someone else's machine. Serve yourself, one request, stop.

Save as `http_timeout_note.py` (the good shape again, with an explicit timeout on the client):

```
# http_timeout_note.py
from http.server import BaseHTTPRequestHandler, HTTPServer
from threading import Thread
from urllib.request import urlopen


class DeskHandler(BaseHTTPRequestHandler):
    def do_GET(self) -> None:
        body = b"desk is open"
        self.send_response(200)
        self.send_header("Content-Type", "text/plain; charset=utf-8")
        self.send_header("Content-Length", str(len(body)))
        self.end_headers()
        self.wfile.write(body)

    def log_message(self, format: str, *args: object) -> None:
        return


def main() -> None:
    server = HTTPServer(("127.0.0.1", 0), DeskHandler)
    thread = Thread(target=server.serve_forever, daemon=True)
    thread.start()
    host, port = server.server_address
    with urlopen(f"http://{host}:{port}/", timeout=2) as resp:
        print(resp.status)
    server.shutdown()
    thread.join(timeout=2)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python http_timeout_note.py
```

Output:

```
200
```

`timeout=` is how a stuck peer fails instead of hanging the process.

The boring rule

- Bind `127.0.0.1` and port `0` in tests.
- Thread + one request + `shutdown` + `join`. The process must exit.
- Use `urllib.request` until you have a reason for a third-party client.
- Catch `HTTPError` and `URLError` at the edge.
- Do not use public echo services as fixtures.

Try this

1. In `desk_http.py`, change the body to `b"shift closed"` and print `resp.status` as well.
2. In `post_ticket.py`, `json.loads` the stored body and print `table`.
3. In `http_missing.py`, send 200 for `/` and 404 for anything else (`self.path`).

Logging and Observability

Logging and Observability

The boring default is the `logging` module: named loggers, levels, and a format you set once. `print` is for programs whose whole job is to print. Production desks log.

Mental model

A **logger** has a name (`"desk"`, `"desk.tickets"`). A **level** filters: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`. A **handler** writes records (usually stderr or a file). A **formatter** turns a record into a line.

`logging.getLogger("desk")` is the same logger everywhere in the process. Configure it at the edge (`main`), not inside library functions.

Observability here means: you can tell what the process did after the fact without attaching a debugger. Levels and a stable format get you most of that. Traces and metrics are extra tools; they are not a reason to skip logs.

Worked examples

Case 1: `basicConfig` and a named logger

Save as `desk_log.py`. Set the format once. Log on the named logger, not the root, so you can turn `desk` up or down later.

```
# desk_log.py
import logging
import sys

logging.basicConfig(
    level=logging.INFO,
    format="%(levelname)s %(name)s: %(message)s",
    stream=sys.stdout,
)
log = logging.getLogger("desk")


def main() -> None:
    log.info("shift opened")
    log.warning("table 0 is invalid")
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_log.py
```

Output:

```
INFO desk: shift opened
WARNING desk: table 0 is invalid
```

`stream=sys.stdout` is for this book's captured output. At work, the default stderr is fine.

Case 2: Child loggers keep the tree

Save as `ticket_log.py`. `desk.tickets` inherits the parent's level unless you change it.

```
# ticket_log.py
import logging
import sys

logging.basicConfig(
    level=logging.INFO,
    format="%(levelname)s %(name)s: %(message)s",
    stream=sys.stdout,
)
log = logging.getLogger("desk.tickets")


def open_ticket(ticket_id: int, table: int) -> None:
    log.info("opened ticket %s at table %s", ticket_id, table)


def main() -> None:
    open_ticket(7, 12)
    log.debug("not printed at INFO")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_log.py
```

Output:

```
INFO desk.tickets: opened ticket 7 at table 12
```

The `debug` line is silent. `%s` lazy-formats only if the line is emitted.

Case 3: JSON-ish lines

Save as `json_log.py`. A custom `Formatter` is enough for log processors that want one object per line. You do not need a framework for this.

```
# json_log.py
import json
import logging
import sys


class JsonIshFormatter(logging.Formatter):
    def format(self, record: logging.LogRecord) -> str:
        return json.dumps(
            {
                "level": record.levelname,
                "logger": record.name,
                "msg": record.getMessage(),
            }
        )


def main() -> None:
    log = logging.getLogger("desk")
    log.setLevel(logging.INFO)
    log.handlers.clear()
    log.propagate = False
    handler = logging.StreamHandler(sys.stdout)
    handler.setFormatter(JsonIshFormatter())
    log.addHandler(handler)
    log.info("shift opened")
    log.warning("table 0 is invalid")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python json_log.py
```

Output:

```
{"level": "INFO", "logger": "desk", "msg": "shift opened"}
{"level": "WARNING", "logger": "desk", "msg": "table 0 is invalid"}
```

`propagate = False` stops a second copy on the root logger.

Case 4: Exceptions belong in the log

Save as `log_exception.py`. `log.exception` records the traceback at `ERROR` after you catch.

```
# log_exception.py
import logging
import sys

logging.basicConfig(
    level=logging.INFO,
    format="%(levelname)s %(name)s: %(message)s",
    stream=sys.stdout,
)
log = logging.getLogger("desk")


def open_table(n: int) -> None:
    if n <= 0:
        raise ValueError(f"table {n}: number must be positive")


def main() -> None:
    try:
        open_table(0)
    except ValueError:
        log.exception("could not open table")
        print("handled")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python log_exception.py
```

Output:

```
ERROR desk: could not open table
Traceback (most recent call last):
  File "log_exception.py", line 20, in main
    open_table(0)
    ~~~~~~~~~~^^^
  File "log_exception.py", line 15, in open_table
    raise ValueError(f"table {n}: number must be positive")
ValueError: table 0: number must be positive
handled
```

The process exits zero because the exception was caught. Line numbers follow the file you saved.

The trap

`print` looks like logging until you need a level, a name, or a file.

Save as `print_as_log.py`:

```
# print_as_log.py
def main() -> None:
    print("shift opened")
    print("table 0 is invalid")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python print_as_log.py
```

Output:

```
shift opened
table 0 is invalid
```

There is no `WARNING`, no logger name, and no way to silence it in production without editing the function. Case 1 is the fix. Keep `print` for CLIs that write the user's answer to stdout.

The boring rule

- `getLogger(__name__)` in library modules. Configure handlers in `main`.
- Log with `%s` (or `{` style if you set it), not `f"..."` built before the level check unless the string is cheap.
- `log.exception` in `except` blocks you handle.

- One format for the process. JSON-ish if a collector wants objects; otherwise the `basicConfig` line format.
- Do not log secrets (API keys, raw card numbers).

Try this

1. In `desk_log.py`, add `log.debug("checking tables")` and rerun at `INFO`, then set `level=logging.DEBUG`.
2. In `ticket_log.py`, log an `error` when `table <= 0`.
3. In `json_log.py`, add `"ticket": 7` only when the message is about a ticket (pass `extra=` or put it in the message).

subprocess and os

subprocess and os

The boring way to run another program is **`subprocess.run`** with a **list of arguments**, `capture_output=True`, and `check=True`. The interpreter you want is **`sys.executable`**. `shell=True` is not the default. `os.environ` is how you read the process environment.

Mental model

`subprocess.run` starts a child, waits, and returns a `CompletedProcess`. `capture_output=True` fills `stdout` and `stderr`. `text=True` gives strings instead of bytes. `check=True` raises `CalledProcessError` on a non-zero exit.

A list `["echo", "7"]` is one program and one argument. A shell string `"echo 7; echo pwned"` is a tiny program in another language. Do not interpolate desk input into that language.

`os.environ` is a mapping of environment variables. It is the right place for `DESK_SHIFT`. It is not a config file.

Worked examples

Case 1: Run this Python, capture stdout

Save as `run_label.py`. `sys.executable` is the interpreter that is already running the file (the same one `uv run python` selected).

```
# run_label.py
import subprocess
import sys


def main() -> None:
    result = subprocess.run(
        [sys.executable, "-c", "print('ticket 7')"],
        capture_output=True,
        text=True,
        check=True,
    )
    print(result.stdout, end="")
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python run_label.py
```

Output:

```
ticket 7
```

From a shell you can also write `uv run python -c "print('ticket 7')"`. Inside Python, prefer `sys.executable` so you do not depend on `PATH`.

Case 2: `check=True` surfaces a failed child

Save as `run_fail.py`. Exit 2 is a failure. Catch `CalledProcessError` when you intend to handle it.

```
# run_fail.py
import subprocess
import sys


def main() -> None:
    try:
        subprocess.run(
            [sys.executable, "-c", "raise SystemExit(2)"],
            capture_output=True,
            text=True,
            check=True,
        )
    except subprocess.CalledProcessError as exc:
        print(type(exc).__name__)
        print(exc.returncode)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python run_fail.py
```

Output:

```
CalledProcessError
2
```

Without the `try`, the traceback is the API: the parent fails too.

Case 3: Read the environment

Save as `shift_env.py`. `os.environ.get` with a default is the boring lookup. Missing keys are not exceptions unless you want them to be (`os.environ["DESK_SHIFT"]`).

```
# shift_env.py
import os


def main() -> None:
    shift = os.environ.get("DESK_SHIFT", "day")
    print(shift)
    print(os.environ.get("NOT_A_DESK_KEY", "missing"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shift_env.py
```

Output:

```
day
missing
```

If you already exported `DESK_SHIFT`, that value prints instead of `day`. Unset it to match this listing.

Case 4: Pass env into the child

Save as `child_env.py`. `env=` replaces the child's environment. Copy `os.environ` first if the child still needs `PATH`.

```
# child_env.py
import os
import subprocess
import sys
```

```
def main() -> None:
    env = os.environ.copy()
    env["DESK_SHIFT"] = "night"
    result = subprocess.run(
        [sys.executable, "-c", "import os; print(os.environ['DESK_SHIFT'])"],
        capture_output=True,
        text=True,
        check=True,
        env=env,
    )
    print(result.stdout, end="")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python child_env.py
```

Output:

```
night
```

The trap

`shell=True` plus an interpolated string runs extra commands. A list does not.

Save as `shell_trap.py`:

```
# shell_trap.py
import subprocess


def main() -> None:
    ticket = "7; echo pwned"
    unsafe = subprocess.run(
        f"echo {ticket}",
        shell=True,
        capture_output=True,
        text=True,
        check=True,
    )
    safe = subprocess.run(
        ["echo", ticket],
        capture_output=True,
```

```
        text=True,
        check=True,
    )
    print("unsafe:", repr(unsafe.stdout))
    print("safe:", repr(safe.stdout))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python shell_trap.py
```

Output:

```
unsafe: '7\npwned\n'
safe: '7; echo pwned\n'
```

The "safe" line is one argument that happens to contain a semicolon. The shell never sees it. Default to the list form. If you truly need a shell, you need a written reason.

The boring rule

- `subprocess.run([...], capture_output=True, text=True, check=True)`.
- Child Python is `sys.executable` (or `uv run python -c ...` from the terminal).
- Never `shell=True` as the default. Never format user input into a shell string.
- Read config-like values from `os.environ`. Do not mutate `os.environ` globally if you can pass `env=` to one child.
- Files still belong to `pathlib`. `os` here is environment and process, not `os.path`.

Try this

1. In `run_label.py`, print `result.returncode` as well (it should be `0`).
2. In `run_fail.py`, drop `check=True` and print `returncode` without a `try`.
3. Run `DESK_SHIFT=swing uv run python shift_env.py` and confirm the first line changes.

Shipping

Static Analysis and Security

Static Analysis and Security

The boring shipping bar is: **`ruff check` is clean**, **dependencies are audited**, and **secrets are not in source**. A linter will not make the desk secure. It will catch the mistakes that keep recurring.

Mental model

Static analysis reads the code without running it. `ruff check` finds unused imports, undefined names, and a pile of foot-guns. Run it in CI the same way you run tests.

Audit means you look up known vulnerabilities in the lockfile. This book's toolchain command is `uv audit`. The older standalone tool is `pip-audit`. Both need a lockfile or an environment worth scanning. They need the network. They are not a substitute for `ruff`.

Secrets are values that grant access: API tokens, passwords, private keys. They live in the environment or a secret store. They do not live in `.py` files, not even "just for now."

Worked examples

Case 1: A tiny program worth checking

Save as `label.py`. This is the clean baseline.

```
# label.py
def label(ticket_id: int, table: int) -> str:
    return f"ticket {ticket_id} → table {table}"


def main() -> None:
    print(label(7, 12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python label.py
```

Output:

```
ticket 7 → table 12
```

Then:

```
uv run ruff check label.py
```

On a clean file ruff prints nothing and exits zero. That silence is the success signal.

Case 2: `ruff` catches an unused import

Save as `unused_import.py`. The program still runs. ruff still fails the check.

```
# unused_import.py
import os


def label(ticket_id: int, table: int) -> str:
    return f"ticket {ticket_id} → table {table}"


def main() -> None:
    print(label(7, 12))


if __name__ == "__main__":
    main()
```

Run the program:

```
uv run python unused_import.py
```

Output:

```
ticket 7 → table 12
```

Run ruff:

```
uv run ruff check unused_import.py
```

Output:

```
F401 [*] `os` imported but unused
 --> unused_import.py:2:8
  |
1 | # unused_import.py
2 | import os
```

```
  |        ^^
help: Remove unused import: `os`
  |
1 | # unused_import.py
  - import os
2 |
  |

Found 1 error.
[*] 1 fixable with the `--fix` option.
```

`ruff check --fix unused_import.py` removes the import. Prefer understanding the diagnostic once, then let `--fix` take the obvious ones.

Case 3: Secrets from the environment

Save as `env_key.py`. Missing key is a hard exit, not a default token in source.

```python
# env_key.py
import os
import sys


def main() -> None:
    key = os.environ.get("DESK_API_KEY")
    if key is None:
        print("DESK_API_KEY is missing")
        sys.exit(1)
    print("key length:", len(key))


if __name__ == "__main__":
    main()
```

Run (unset):

```
uv run python env_key.py
```

Output:

```
DESK_API_KEY is missing
```

Run with a value:

```
DESK_API_KEY=abc uv run python env_key.py
```

Output:

```
key length: 3
```

Never print the key itself.

Case 4: Audit commands (project directory)

In a project that has `pyproject.toml` and a lockfile:

```
uv lock
uv audit
```

`uv audit` talks to a vulnerability service (OSV by default). It is allowed to need the network. If you maintain an older pipeline, `pip-audit` against the same lockfile is the same idea.

Do not paste audit JSON into the book as a fixture: the database changes. Wire the command into CI and fail the build on known issues you have not ignored on purpose (`uv audit --ignore ...` is explicit).

The trap

A literal in the file "works" and leaks the first time the repo is copied.

Save as `leaked_key.py`:

```
# leaked_key.py
API_KEY = "not-a-real-key"


def main() -> None:
    print("would send", API_KEY)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python leaked_key.py
```

Output:

```
would send not-a-real-key
```

Git history keeps it after you delete the line. Case 3 is the fix: read `DESK_API_KEY` from the environment. Rotate anything that ever sat in source.

The boring rule

- `uv run ruff check` (and `ruff format`) on every change.
- `uv lock` then `uv audit` (or `pip-audit`) in CI.
- Secrets in the environment or a vault. Never in source, tests, or example configs you commit.
- Do not disable a ruff rule globally to silence one line; use a targeted `noqa` with a reason, rarely.
- Treat "the program ran" as necessary, not sufficient, to ship.

Try this

1. Add an unused `import json` to `label.py` and run `uv run ruff check label.py`.
2. Run `ruff check --fix unused_import.py` and confirm the import is gone.
3. In `env_key.py`, reject an empty string (`key` is `""`) the same way you reject `None`.

Packaging and Resources

Packaging and Resources

A shippable library is a **package** with a `pyproject.toml`, not a pile of scripts. Data files travel with the package and are read through `importlib.resources`, not `open("banner.txt")` relative to whoever's current directory.

Mental model

`[project]` in `pyproject.toml` is the public name, version, and description. `[build-system]` names the backend. **hatchling** is a boring default; `uv build` is how this book builds.

A **package** is a directory with `__init__.py` (or a namespace you chose on purpose). Import `deskpkg`, do not hope `banner.txt` is next to the shell.

`importlib.resources.files("deskpkg")` is a traversable of files inside that package, whether it lives as a directory or inside a wheel. `read_text` still takes `encoding="utf-8"`.

Worked examples

Case 1: The package layout (inline)

Save these three files as a tiny project. You do not have to install it to learn the shape.

`pyproject.toml`:

```toml
[project]
name = "deskpkg"
version = "1.0.0"
description = "Desk labels and a banner."
requires-python = ">=3.14"
readme = "README.md"

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

`deskpkg/__init__.py`:

```python
# deskpkg/__init__.py
__version__ = "1.0.0"
```

```
def label(ticket_id: int, table: int) -> str:
    return f"ticket {ticket_id} → table {table}"
```

`deskpkg/banner.txt`:

```
desk is open
```

Hatchling includes package data next to Python modules by default for this layout. If a backend does not, you declare the files in `pyproject.toml` instead of hoping.

Build (from that project directory):

```
uv build
```

That writes a wheel and an sdist under `dist/`. Installing is `uv pip install dist/deskpkg-1.0.0-py3-none-any.` (the exact wheel name follows the version).

Case 2: Read a resource without a cwd guess

Save as `read_banner.py`. This listing **creates** the package in a temporary directory so it runs as a complete program. Production code imports `deskpkg` after install and skips the `sys.path` dance.

```
# read_banner.py
import importlib.resources
import sys
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        root = Path(raw)
        pkg = root / "deskpkg"
        pkg.mkdir()
        (pkg / "__init__.py").write_text(
            '__version__ = "1.0.0"\n', encoding="utf-8"
        )
        (pkg / "banner.txt").write_text("desk is open\n", encoding="utf-8")
        sys.path.insert(0, str(root))
        import deskpkg

        banner = (
            importlib.resources.files("deskpkg")
            .joinpath("banner.txt")
            .read_text(encoding="utf-8")
        )
```

```
        print(deskpkg.__version__)
        print(banner, end="")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python read_banner.py
```

Output:

```
1.0.0
desk is open
```

Case 3: Call the public function after import

Save as `use_label.py`. Same temp package, no banner this time.

```
# use_label.py
import sys
from pathlib import Path
from tempfile import TemporaryDirectory


def main() -> None:
    with TemporaryDirectory() as raw:
        root = Path(raw)
        pkg = root / "deskpkg"
        pkg.mkdir()
        (pkg / "__init__.py").write_text(
            "def label(ticket_id, table):\n"
            '    return f"ticket {ticket_id} → table {table}"\n',
            encoding="utf-8",
        )
        sys.path.insert(0, str(root))
        from deskpkg import label

        print(label(7, 12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python use_label.py
```

Output:

```
ticket 7 → table 12
```

The trap

Opening a data file by name looks fine from the directory where you wrote it.

Save as `cwd_banner.py`:

```
# cwd_banner.py
from pathlib import Path


def main() -> None:
    path = Path("banner.txt")
    try:
        print(path.read_text(encoding="utf-8"), end="")
    except FileNotFoundError:
        print("banner.txt is not in the current directory")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python cwd_banner.py
```

Output:

```
banner.txt is not in the current directory
```

A user who runs `uv run python /path/to/cwd_banner.py` from `$HOME` will hit this every time. `importlib.resources` is the fix.

The boring rule

- One `pyproject.toml` with `[project]` and a declared `[build-system]` (hatchling is enough).
- Put library code in a package directory. Keep `__version__` in one place.
- Read bundled files with `importlib.resources`, not `Path("filename")`.
- Build with `uv build`. Do not hand-roll `setup.py` for a new project.
- The current working directory is not part of your package's API.

Try this

1. In `read_banner.py`, add `deskpkg/label.txt` with `ticket 7` and print it the same way as the banner.
2. Add a `README.md` one-liner to the inline project in Case 1 (the `[project] readme` field names it).
3. In `use_label.py`, also print `deskpkg.__name__`.

Building and Releasing

Building and Releasing

A release is a **version in `pyproject.toml`**, a **wheel** (and usually an sdist) from **`uv build`**, and a tag that matches. The code that runs after install should print that same version. Do not have three version strings.

Mental model

Version is `[project].version` (or a dynamic scheme you chose on purpose). Bump it when you publish, not when you save a file.

`uv build` asks the build backend (hatchling here) for two artifacts:

- **sdist** — source tree in a tarball (`deskshift-1.2.0.tar.gz`)
- **wheel** — installable zip (`deskshift-1.2.0-py3-none-any.whl`)

Wheels are what `uv pip install` / `uv add` consume from an index. The `py3-none-any` tags mean "pure Python, any OS." That is the boring target until you wrap a C library.

Releasing is: tests green, ruff clean, version bumped, `uv build`, upload to your index, git tag `v1.2.0`. This chapter shows the files and the build command, not a hosting vendor.

Worked examples

Case 1: Version lives in `pyproject.toml`

Save as a project:

`pyproject.toml`:

```toml
[project]
name = "deskshift"
version = "1.2.0"
description = "Shift labels for a small desk."
requires-python = ">=3.14"

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

`deskshift/__init__.py`:

```
# deskshift/__init__.py
__version__ = "1.2.0"


def shift_label(name: str) -> str:
    return f"shift {name}"
```

The duplicated `"1.2.0"` is the thing you will eventually replace with `importlib.metadata.version("deskshift")` **after** the package is installed. Until then, one constant next to the function is acceptable if you bump both in the same commit.

Case 2: A complete module that prints the version

Save as `show_version.py` and run it without installing. This is the same string you put in `pyproject.toml`.

```
# show_version.py
__version__ = "1.2.0"


def shift_label(name: str) -> str:
    return f"shift {name}"


def main() -> None:
    print(__version__)
    print(shift_label("night"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python show_version.py
```

Output:

```
1.2.0
shift night
```

Case 3: Build commands

From the project directory that contains `pyproject.toml`:

```
uv build
```

You should see files under `dist/`:

```
dist/deskshift-1.2.0.tar.gz
dist/deskshift-1.2.0-py3-none-any.whl
```

Install the wheel into an isolated check:

```
uv pip install dist/deskshift-1.2.0-py3-none-any.whl
uv run python -c "import deskshift; print(deskshift.__version__)"
```

Expected output of that `-c`:

```
1.2.0
```

If the wheel name differs, list `dist/` and install the file that is there. Do not edit a wheel by hand.

Case 4: Read the installed version

Save as `installed_version.py`. `importlib.metadata.version` looks at the installed distribution. In this listing we fall back to a constant so the file still runs when you have not installed `deskshift`.

```
# installed_version.py
from importlib.metadata import PackageNotFoundError, version

__version__ = "1.2.0"


def package_version() -> str:
    try:
        return version("deskshift")
    except PackageNotFoundError:
        return __version__


def main() -> None:
    print(package_version())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python installed_version.py
```

Output:

```
1.2.0
```

After a real install, the metadata path wins. That is the boring long-term source of truth.

The trap

A comment, a constant, and `pyproject.toml` that disagree.

Save as `wrong_version.py`:

```
# wrong_version.py
# version 1.1.0  - leftover comment
__version__ = "1.2.0"


def main() -> None:
    print("package 1.0.0")
    print(__version__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python wrong_version.py
```

Output:

```
package 1.0.0
1.2.0
```

Three numbers, one process. Support will ask "which version?" and every answer will be true. Print `__version__` (or `importlib.metadata.version`) only. Delete the comment. Bump `pyproject.toml` in the same change.

The boring rule

- One version per release, in `pyproject.toml`, tagged in git as `vX.Y.Z`.
- `uv build` produces sdist + wheel. Ship both unless you have a reason not to.
- Pure Python wheels (`py3-none-any`) until you truly need compiled code.
- After install, prefer `importlib.metadata.version("deskshift")`.
- Do not release from a dirty tree. Tests and `ruff check` first.

Try this

1. Change `__version__` in `show_version.py` to `1.2.1` and print `shift_label("day")`.
2. Add `authors = [{name = "Desk"}]` to the Case 1 `pyproject.toml`.
3. In `installed_version.py`, print a prefix `deskshift` before the version.

Documentation and APIs

Documentation and APIs

The boring public API is a **small set of functions with docstrings**, plus a CLI that calls them. `argparse` is enough. The docstring is the contract you can keep.

Mental model

An **API** is what callers are allowed to depend on: names, arguments, return values, and exceptions. Everything else is private, even if Python will still import it.

A **docstring** is the string literal under the `def` (or on the module). `help()` and `inspect.getdoc` read it. Write one paragraph that says what the function returns and what it raises.

argparse parses `sys.argv` into those same arguments. The CLI is a user of the function, not a second implementation.

Worked examples

Case 1: A docstring is the promise

Save as `ticket_api.py`. The function is the API. `main` only prints.

```
# ticket_api.py
def label(ticket_id: int, table: int) -> str:
    """Return a one-line ticket label for the desk display."""
    return f"ticket {ticket_id} → table {table}"


def main() -> None:
    print(label.__doc__)
    print(label(7, 12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_api.py
```

Output:

```
Return a one-line ticket label for the desk display.
ticket 7 → table 12
```

Case 2: argparse wraps the same function

Save as `label.py`. Flags map onto `label`. There is no copy of the format string in `main`.

```python
# label.py
import argparse


def build_parser() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser(description="Print a ticket label.")
    parser.add_argument("ticket_id", type=int)
    parser.add_argument("--table", type=int, required=True)
    return parser


def label(ticket_id: int, table: int) -> str:
    """Return a one-line ticket label for the desk display."""
    return f"ticket {ticket_id} → table {table}"


def main() -> None:
    args = build_parser().parse_args()
    print(label(args.ticket_id, args.table))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python label.py 7 --table 12
```

Output:

```
ticket 7 → table 12
```

Case 3: `-h` is documentation too

Same file, different argv.

```
uv run python label.py -h
```

Output:

```
usage: label.py [-h] --table TABLE ticket_id

Print a ticket label.

positional arguments:
  ticket_id

options:
  -h, --help     show this help message and exit
  --table TABLE
```

That help text is why `description=` exists. Keep it accurate.

Case 4: Invalid input fails at the edge

Save as `label_bad.py` (same parser, explicit `parse_args` on a list so the listing stays non-interactive).

```python
# label_bad.py
import argparse


def build_parser() -> argparse.ArgumentParser:
    parser = argparse.ArgumentParser(description="Print a ticket label.")
    parser.add_argument("ticket_id", type=int)
    parser.add_argument("--table", type=int, required=True)
    return parser


def main() -> None:
    parser = build_parser()
    try:
        parser.parse_args(["nope", "--table", "12"])
    except SystemExit as exc:
        print("exit", exc.code)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python label_bad.py
```

Output (stderr from argparse, then stdout):

```
usage: label_bad.py [-h] --table TABLE ticket_id
```

```
label_bad.py: error: argument ticket_id: invalid int value: 'nope'
exit 2
```

Callers of `label()` still pass an `int`. The CLI is the one that translates strings.

The trap

A CLI that formats the string itself, and a function that formats it differently.

Save as `two_apis.py`:

```python
# two_apis.py
def label(ticket_id: int, table: int) -> str:
    """Return a one-line ticket label for the desk display."""
    return f"ticket {ticket_id} → table {table}"


def main() -> None:
    ticket_id = 7
    table = 12
    print(f"#{ticket_id} @ {table}")
    print(label(ticket_id, table))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python two_apis.py
```

Output:

```
#7 @ 12
ticket 7 → table 12
```

Two formats, one desk. Tests will pick one and production the other. `main` should only `print(label(...))`.

The boring rule

- Document the public functions. Keep the set small.
- CLI via `argparse` (or a thin wrapper) that calls those functions.
- One format string, one place.
- Raise `ValueError` / `TypeError` in the library. Let argparse handle argv mistakes.
- Do not promise a plugin system in the docstring of a 12-line module.

Try this

1. In `ticket_api.py`, raise `ValueError` when `table <= 0` and mention it in the docstring.
2. In `label.py`, add `--status` with default `"open"` and include it in `label`.
3. Run `uv run python label.py 7` without `--table` and read the argparse error.

Long Term

inspect and Reflection

inspect and Reflection

`inspect` and `getattr` are **last resorts**: useful for debugging, adapters, and tests that need a signature. They are not how the desk opens a ticket on a normal path. Call the function. Name the attribute.

Mental model

Reflection means asking a live object what it is: its name, its signature, its attributes. `inspect.signature(fn)` is the callable's parameters. `getattr(obj, "table")` is `obj.table` with a string name.

When the name is a string you computed, you have lost the editor, the type checker, and most grep. That cost is acceptable in a test helper or a serializer. It is not acceptable in `open_table`.

Worked examples

Case 1: `inspect.signature`

Save as `show_sig.py`. This is what you print when a helper's arguments drifted.

```
# show_sig.py
import inspect


def open_table(n: int, *, reason: str = "shift") -> str:
    return f"opened table {n} ({reason})"


def main() -> None:
    sig = inspect.signature(open_table)
    print(sig)
    print(list(sig.parameters))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python show_sig.py
```

Output:

```
(n: int, *, reason: str = 'shift') -> str
['n', 'reason']
```

Case 2: `getattr` with a default

Save as `ticket_getattr.py`. The default argument is the only reason to prefer `getattr` over `obj.table` here.

```
# ticket_getattr.py
class Ticket:
    def __init__(self, ticket_id: int, table: int) -> None:
        self.id = ticket_id
        self.table = table


def main() -> None:
    ticket = Ticket(7, 12)
    print(getattr(ticket, "table"))
    print(getattr(ticket, "status", "open"))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_getattr.py
```

Output:

```
12
open
```

`ticket.table` would have been clearer for the first line. The second line is the case `getattr` is for: optional attribute, default if missing.

Case 3: `inspect.getdoc`

Save as `show_doc.py`. Same string you wrote under `def`.

```
# show_doc.py
import inspect


def label(ticket_id: int, table: int) -> str:
    """Return a one-line ticket label for the desk display."""
    return f"ticket {ticket_id} → table {table}"


def main() -> None:
    print(inspect.getdoc(label))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python show_doc.py
```

Output:

```
Return a one-line ticket label for the desk display.
```

The trap

A tiny dispatcher that looks up functions by string. It runs. It also hides every call site.

Save as `dispatch_trap.py`:

```
# dispatch_trap.py
def open_ticket() -> str:
    return "opened ticket"


def close_ticket() -> str:
    return "closed ticket"


def main() -> None:
    name = "open_ticket"
    fn = globals().get(name)
    if fn is None:
        raise KeyError(name)
    print(fn())
```

```
if __name__ == "__main__":
    main()
```

Run:

```
uv run python dispatch_trap.py
```

Output:

```
opened ticket
```

The boring version is `print(open_ticket())`. If you have two commands, use `if name == "open":` or a dict **you write as a dict**, not `globals()`. `getattr(module, user_input)` is the same trap with extra extra steps.

Fix:

```
# dispatch_fix.py
def open_ticket() -> str:
    return "opened ticket"


def close_ticket() -> str:
    return "closed ticket"


def main() -> None:
    commands = {
        "open": open_ticket,
        "close": close_ticket,
    }
    print(commands["open"]())


if __name__ == "__main__":
    main()
```

Run:

```
uv run python dispatch_fix.py
```

Output:

```
opened ticket
```

The keys are the API. Unknown names raise `KeyError` on the map, not on the module guts.

The boring rule

- Call functions by name. Read attributes by name.
- `inspect.signature` / `getdoc` for tools, docs, and tests — not for the happy path.
- `getattr(obj, name, default)` when the name is actually dynamic.
- Never `getattr` on user input against a module. Use an explicit dict.
- If a plugin protocol needs reflection, keep it in one adapter file.

Try this

1. In `show_sig.py`, print `sig.parameters["reason"].default`.
2. In `ticket_getattr.py`, use `ticket.table` instead of `getattr` for the first print.
3. In `dispatch_fix.py`, look up `"pay"` and catch `KeyError`.

ctypes and FFI

ctypes and FFI

Avoid FFI unless you must. Python's standard types, plus a subprocess or a well-tested package, cover almost all desk work. `ctypes` talks to C in-process. That is a sharp edge: crashes are not `except Exception`.

Mental model

FFI (foreign function interface) means calling code compiled for another language, usually C, from Python. `ctypes` ships with Python. You describe C types (`c_int`, `c_char_p`) and call a shared library.

`ctypes.sizeof` reports the C size of a type. `ctypes.c_int` is a mutable box around a C `int`. On common platforms that size is 4.

Calling `libc` (`strlen`, and friends) is **not** portable as a copy-paste: the library name is `libc.so.6` on many Linux systems, `libSystem` on macOS, and something else on Windows. This book does not make that the default example.

If you need C, prefer a maintained package with wheels, or a small extension you build on purpose — not ad-hoc `ctypes` in a ticket handler.

Worked examples

Case 1: `c_int` and `sizeof` (no extra C files)

Save as `c_int_size.py`. This runs anywhere ctypes runs.

```
# c_int_size.py
import ctypes


def main() -> None:
    table = ctypes.c_int(12)
    print(table.value)
    print(ctypes.sizeof(table))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python c_int_size.py
```

Output:

```
12
4
```

`table.value` is the Python `int`. Changing `.value` mutates the box. A plain `int` is enough until a C API demands this object.

Case 2: A tiny C-like struct

Save as `ticket_struct.py`. `_fields_` is the C layout. Still no `.c` file.

```
# ticket_struct.py
import ctypes


class Ticket(ctypes.Structure):
    _fields_ = [
        ("id", ctypes.c_int),
        ("table", ctypes.c_int),
    ]


def main() -> None:
    ticket = Ticket(7, 12)
    print(ticket.id, ticket.table)
    print(ctypes.sizeof(ticket))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_struct.py
```

Output:

```
7 12
8
```

Two `c_int` fields, 4 + 4, no padding on this layout. Do not depend on padding guesses when you talk to a real C header — match the header.

Case 3: The Python you wanted instead

Save as `ticket_python.py`. Same data, no FFI.

```
# ticket_python.py
from dataclasses import dataclass


@dataclass
class Ticket:
    id: int
    table: int


def main() -> None:
    ticket = Ticket(7, 12)
    print(ticket.id, ticket.table)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python ticket_python.py
```

Output:

```
7 12
```

This is the default. Case 2 exists so you recognize ctypes when a library forces it.

The trap

Loading libc "because we can" for a job `len` already does.

Save as `skip_libc.py`. We do **not** call `CDLL`. We print why the boring path wins.

```
# skip_libc.py
def main() -> None:
    text = "desk"
    print(len(text))
    print("no libc")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python skip_libc.py
```

Output:

```
4
no libc
```

A `strlen` call through `ctypes.CDLL("libc.so.6")` may work on your Linux box and fail on the next one. `len` is Unicode-aware in Python; `strlen` is bytes and a terminator. They are not the same function.

If you truly must call libc, isolate it behind one function, pass `bytes`, and test on every OS you ship. That is a last resort, not a style.

The boring rule

- Do not use ctypes for business logic.
- Prefer a dataclass, `array`, or `memoryview` over a `Structure`.
- Prefer a wheel from PyPI over a handwritten `CDLL`.
- Prefer `subprocess` plus a CLI over in-process C if the boundary is already a process.
- Never let FFI exceptions look like Python bugs: wrap the call, validate pointers, and keep the surface tiny.

Try this

1. In `c_int_size.py`, set `table.value = 3` and print it again.
2. In `ticket_struct.py`, add a `status` field as `c_int` (`0` open, `1` paid) and print it.
3. Rewrite Case 2's print using the dataclass from Case 3 — no ctypes.

Performance Tuning

Performance Tuning

Measure first. The boring tools are `timeit` for a small expression and `cProfile` for "where did the time go." Do not rewrite a ticket loop because it feels slow.

Mental model

CPU time and wall time are different. `timeit` repeats a snippet to damp noise. `cProfile` records function calls in one run. Both answer questions. Neither is a reason to add a cache on day one.

The desk is usually waiting on disk, the network, or a human. Faster JSON parsing will not fix that. Profile, then change the hot function, then measure again.

Timings themselves are not stable across machines. The listings below print **results** and the fact that a profiler saw the function — not a number of seconds you would treat as a fixture.

Worked examples

Case 1: Get the answer right, then time it

Save as `count_open.py`. `timeit.timeit` runs `count_open` 1000 times. We print the count, then a marker that the timer finished.

```
# count_open.py
import timeit


def count_open() -> int:
    tickets = [{"status": "open"}] * 100 + [{"status": "paid"}] * 50
    return sum(1 for t in tickets if t["status"] == "open")


def main() -> None:
    print(count_open())
    timeit.timeit(count_open, number=1000)
    print("measured")


if __name__ == "__main__":
    main()
```

Run:

```
uv run python count_open.py
```

Output:

```
100
measured
```

To see seconds on your machine, print the return value of `timeit.timeit`. Do not commit that float as a test.

Case 2: `cProfile` names the function

Save as `profile_open.py`. `runcall` profiles one invocation. We only assert the profiler recorded `count_open`.

```
# profile_open.py
import cProfile


def count_open() -> int:
    tickets = [{"status": "open"}] * 100 + [{"status": "paid"}] * 50
    return sum(1 for t in tickets if t["status"] == "open")


def main() -> None:
    profiler = cProfile.Profile()
    n = profiler.runcall(count_open)
    names = [
        s.code.co_name
        for s in profiler.getstats()
        if hasattr(s.code, "co_name")
    ]
    print(n)
    print("count_open" in names)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python profile_open.py
```

Output:

```
100
True
```

From a terminal, the human-readable table is:

```
uv run python -m cProfile -s tottime count_open.py
```

Sort by `tottime`. Read the top rows. Ignore the rest until they matter.

Case 3: Two implementations, same answer

Save as `scan_vs_sum.py`. Correctness first. Timing second.

```
# scan_vs_sum.py
def count_loop(tickets: list[dict[str, str]]) -> int:
    n = 0
    for t in tickets:
        if t["status"] == "open":
            n += 1
    return n


def count_sum(tickets: list[dict[str, str]]) -> int:
    return sum(1 for t in tickets if t["status"] == "open")


def main() -> None:
    tickets = [{"status": "open"}] * 100 + [{"status": "paid"}] * 50
    a = count_loop(tickets)
    b = count_sum(tickets)
    print(a)
    print(a == b)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python scan_vs_sum.py
```

Output:

```
100
True
```

If both are correct and neither shows up in a profile, keep the one that reads better (`count_sum` here).

The trap

Micro-optimizing a string that is not the bottleneck.

Save as `clever_label.py`:

```
# clever_label.py
def label_parts(ticket_id: int, table: int) -> str:
    return "ticket " + str(ticket_id) + " → table " + str(table)


def label_f(ticket_id: int, table: int) -> str:
    return f"ticket {ticket_id} → table {table}"


def main() -> None:
    print(label_parts(7, 12))
    print(label_f(7, 12))
    print(label_parts(7, 12) == label_f(7, 12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python clever_label.py
```

Output:

```
ticket 7 → table 12
ticket 7 → table 12
True
```

The `+` version is not a win you will measure on a desk. The f-string is the default. Optimize after `cProfile` points at a function that actually runs a million times.

The boring rule

- Make it correct. Then measure. Then change one thing.
- `timeit` for a snippet. `python -m cProfile -s tottime` for a program.
- Do not treat wall-clock floats as golden tests.
- I/O and algorithms beat micro-syntax.
- Keep the readable version when the profile is a tie.

Try this

1. Print `timeit.timeit(count_open, number=1000)` in `count_open.py` and note it changes run to run.
2. Add a third function that uses a list comprehension and check it against `count_sum`.
3. Run `uv run python -m cProfile -s tottime profile_open.py` and find `count_open` in the table.

Designing for the Long Term

Designing for the Long Term

Software lasts when the **boundaries are real**, the **API is small**, and the defaults stay boring: functions, dataclasses, the standard library, tests, and a version you can explain. This chapter is a recap you can keep on the desk.

Mental model

A **package boundary** is a directory other code imports. Inside, you may shuffle files. Across it, names are promises.

A **stable API** is a handful of functions (and maybe a dataclass) with docstrings. Hidden helpers start with `_` or live in a module you do not document.

Long-term Python in this book means: Python 3.14, `uv`, `ruff`, `pytest`, UTC in storage, `pathlib`, `logging`, `subprocess.run` as a list, no secrets in git, wheels from `uv build`.

Worked examples

Case 1: Two modules, one public function

Save as a tiny package in a temp tree and import it. Production looks the same without `TemporaryDirectory`.

`desk/__init__.py` exports `label` only. `desk/format_ticket.py` can change.

```
# stable_desk.py
import sys
from pathlib import Path
from tempfile import TemporaryDirectory


INIT = '''\
from desk.format_ticket import label

__all__ = ["label"]
'''

FORMAT = '''\
def label(ticket_id: int, table: int) -> str:
```

```
    """Return a one-line ticket label for the desk display."""
    return f"ticket {ticket_id} → table {table}"
'''


def main() -> None:
    with TemporaryDirectory() as raw:
        root = Path(raw)
        pkg = root / "desk"
        pkg.mkdir()
        (pkg / "__init__.py").write_text(INIT, encoding="utf-8")
        (pkg / "format_ticket.py").write_text(FORMAT, encoding="utf-8")
        sys.path.insert(0, str(root))
        import desk

        print(desk.label(7, 12))
        print(desk.__all__)


if __name__ == "__main__":
    main()
```

Run:

```
uv run python stable_desk.py
```

Output:

```
ticket 7 → table 12
['label']
```

Callers import `desk.label`. They do not import `desk.format_ticket` unless you say so.

Case 2: `pyproject.toml` is the project boundary

The long-term file at the root of a library:

```
[project]
name = "desk"
version = "1.0.0"
description = "Labels and shifts for a small desk."
requires-python = ">=3.14"

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

That file, `desk/`, tests, and `README.md` are the repo. Everything else is extra.

Case 3: One process that still looks like the first chapter

Save as `desk_open.py`. A function, a print, a `main` guard. Types where they help. No framework.

```
# desk_open.py
def open_table(n: int) -> str:
    if n <= 0:
        raise ValueError(f"table {n}: number must be positive")
    return f"opened table {n}"


def main() -> None:
    print(open_table(12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python desk_open.py
```

Output:

```
opened table 12
```

If this is enough, stop. Add a package when a second module would otherwise collide.

The trap

A "platform" on day one: a base class, a plugin registry, and a settings object for a label.

Save as `too_much.py`:

```
# too_much.py
class BaseLabel:
    def render(self, ticket_id: int, table: int) -> str:
        raise NotImplementedError


class DeskLabel(BaseLabel):
    def render(self, ticket_id: int, table: int) -> str:
        return f"ticket {ticket_id} → table {table}"
```

```
def main() -> None:
    print(DeskLabel().render(7, 12))


if __name__ == "__main__":
    main()
```

Run:

```
uv run python too_much.py
```

Output:

```
ticket 7 → table 12
```

The boring package from Case 1 prints the same string with one function. Add a class when **state lives** and **behavior belongs to it**. Add plugins when you have a second implementation, not a slogan.

The boring rule

- Export a small `__all__`. Keep helpers behind it.
- Store UTC, log with `logging`, run children with `subprocess.run([...])`, read files with `Path`.
- Test the public functions. Lint with `ruff`. Audit with `uv audit`.
- Version in `pyproject.toml`. Build with `uv build`.
- Reflection and FFI stay at the edges.
- When in doubt, write the function from Case 3.

Try this

1. In `stable_desk.py`, add `desk/shifts.py` with `shift_label(name)` and export it from `__init__.py`.
2. In `desk_open.py`, catch `ValueError` for `open_table(0)` and print the error.
3. Delete `BaseLabel` from `too_much.py` and keep a single `label` function.

Appendices

Glossary

Glossary

Short definitions for terms this book uses as defaults. Words are defined on first use in the chapters; this list is for lookup.

API. The names, arguments, return values, and exceptions callers may depend on.

argparse. Standard-library parser for `sys.argv`. The CLI should call your functions, not reimplement them.

aware datetime. A `datetime` with `tzinfo` set. Store these (UTC). Contrast **naive datetime**.

`cProfile`. Standard-library profiler. Run `uv run python -m cProfile -s tottime file.py` after the program is correct.

CSV. Row-and-column text. Use `csv` with `newline=""` and UTF-8.

ctypes. Standard-library FFI. Call C types and libraries. Last resort; prefer Python types.

dataclass. Class generated from fields (`@dataclass`). Boring model for tickets and shifts.

docstring. String literal right under `def` or on a module. The written contract.

encoding. How bytes become text. This book always passes `encoding="utf-8"` for text files.

environment variable. A name/value pair on the process (`os.environ`). Where secrets and `DESK_SHIFT` live.

FFI. Foreign function interface: calling C (or similar) from Python. Avoid unless you must.

hatchling. A build backend. Named in `[build-system]` so `uv build` can produce wheels.

HTTPError. `urllib` exception for HTTP status errors such as 404.

`importlib.resources`. Read files shipped inside a package, independent of the current directory.

`inspect`. Standard-library reflection (`signature`, `getdoc`). For tools and tests, not the happy path.

ISO-8601. Timestamp text such as `2026-09-07T14:30:00+00:00`. What you put in JSON.

JSON. Nested text format for APIs and files (`json` module). No native datetime type.

`logging`. Standard-library logs: named loggers, levels, handlers. The boring default over `print` in services.

`main` guard. `if __name__ == "__main__":`. Runs `main` only when the file is the program.

naive datetime. A `datetime` with `tzinfo is None`. Do not store these as the system of record.

package. Importable directory (usually with `__init__.py`). The unit you ship.

path. A location on disk. `pathlib.Path` is the object; `/` joins relative parts.

pip-audit. Standalone vulnerability scanner. This book prefers `uv audit` when you already use `uv`.

pyproject.toml. Project metadata: name, version, dependencies, build backend.

pytest. Test runner. `uv run pytest`.

ruff. Linter and formatter. `uv run ruff check` / `ruff format`.

sdist. Source distribution tarball from `uv build`.

StringIO. In-memory text stream (`io.StringIO`). Same shape as a text file for tests.

subprocess.run. Run a child process. List of args, `capture_output=True`, `check=True`. No `shell=True` by default.

sys.executable. Path of the running interpreter. Use it to spawn Python children.

TemporaryDirectory. Temp folder that is deleted when the `with` block ends.

TextIO. Type of a text stream (`typing.TextIO`): files, `StringIO`.

timedelta. A duration (`datetime.timedelta`), not a clock.

timeit. Repeat a snippet and measure time. Do not freeze the float in tests.

TOML. Config format. `tomllib` **reads** it in the standard library; it does not write.

type hint. Annotation such as `n: int`. Used from the typing part onward where they help.

UTC. The zone you store (`datetime.timezone.utc`). Convert to a desk `ZoneInfo` for display.

uv. Toolchain: install Python, run programs (`uv run python file.py`), lock, build, audit.

wheel. Built install artifact (`*.whl`) from `uv build`.

ZoneInfo. IANA time zone (`zoneinfo.ZoneInfo("America/Chicago")`) for display and local wall times.

Command Cheat Sheet

Command Cheat Sheet

Commands this book actually uses. Run them from the directory that holds the file or the `pyproject.toml`.

Python

Command	What it does
`uv run python file.py`	Run a program with the project's Python 3.14
`uv run python -c "print('desk')"`	Run a one-liner
`uv run python -m cProfile -s tottime file.py`	Profile a program

Inside a program, spawn that same interpreter with `sys.executable`, not a bare `python` on `PATH`.

uv

Command	What it does
`uv lock`	Write/update the lockfile
`uv sync`	Install the locked environment
`uv add pkg`	Add a dependency
`uv build`	Build sdist + wheel into `dist/`
`uv audit`	Scan locked deps for known issues
`uv pip install dist/*.whl`	Install a wheel you just built

ruff

Command	What it does
`uv run ruff check .`	Lint
`uv run ruff check --fix file.py`	Lint and apply safe fixes
`uv run ruff format .`	Format

Silence is success for `ruff check`.

pytest

Command	What it does
`uv run pytest`	Run tests
`uv run pytest -q`	Quieter
`uv run pytest path/to/test_file.py`	One file

Tiny reminders

Run a complete program:

```
uv run python hello.py
```

```
# hello.py
def main() -> None:
    print("desk is open")


if __name__ == "__main__":
    main()
```

```
desk is open
```

Child process (list of args, no shell):

```
# run_child.py
import subprocess
import sys


def main() -> None:
    result = subprocess.run(
        [sys.executable, "-c", "print('ticket 7')"],
        capture_output=True,
        text=True,
        check=True,
    )
    print(result.stdout, end="")


if __name__ == "__main__":
    main()
```

```
uv run python run_child.py
```

```
ticket 7
```

Secrets stay in the environment (`DESK_API_KEY`), never in the file you commit.

Times you store are UTC (`datetime.now(timezone.utc)`), not naive local clocks.

Text files: `Path.read_text(encoding="utf-8")` and `write_text(..., encoding="utf-8")`.