

COMPANION VOLUME • STAGE 01 OF THE ENGINEER'S SYLLABUS

The Engineer's Primer

A primer was the first book a child learned to read from, and it is still the name for the first coat of paint, the layer everything after it holds on to. This volume is both for programming. It takes you from values and expressions to programs that read messy files, keep state, talk to the network and survive being changed. Every idea comes with a program you can run, and it shows what Python is doing underneath, so the rules make sense instead of having to be memorised.

CHAPTERS	CODE EXAMPLES	EXERCISES	STAGE BUDGET
20	20	60	300 h

Contents

How to use this volume	4
Part I · Thinking in code	6
Chapter 1 · Values, types and expressions	7
Chapter 2 · Names, assignment and scope	14
Chapter 3 · Control flow and loop invariants	20
Chapter 4 · Functions	27
Chapter 5 · Recursion and the call stack	34
Part II · Data	41
Chapter 6 · Lists and tuples	42
Chapter 7 · Dictionaries and sets	49
Chapter 8 · Strings, Unicode and UTF-8	56
Chapter 9 · Files, CSV and JSON	62
Part III · Getting it right	69
Chapter 10 · Errors and exceptions	70
Chapter 11 · Testing	77
Chapter 12 · Debugging	84
Part IV · Structure	91
Chapter 13 · Objects and classes	92
Chapter 14 · Composition and interfaces	99
Chapter 15 · References, mutation and aliasing	106
Chapter 16 · Modules, packages and environments	113
Part V · Programs in the world	120
Chapter 17 · Reading unfamiliar code	121
Chapter 18 · Command-line tools	128
Chapter 19 · Programs with state	135
Chapter 20 · Talking to APIs	142
The deliverable	149

The twenty rules 152

Reading list 154

HOW TO USE THIS VOLUME

This is the longest stage, and the one people rush. The goal is not to learn Python's syntax, which takes a few weeks. It's the ability to hold a problem in your head, break it into parts, and write those parts as code that someone else can read and change. Every later stage assumes it. The table shows where each of the stage's fourteen topics is taught here, and what to read or practise alongside.

STAGE 01 TOPIC	CHAPTERS	READ OR PRACTISE ALONGSIDE
Values, types, variables, expressions, statements, scope	Ch. 1–2	Composing Programs 1.2–1.3
Control flow, iteration, and the loop invariants that make it correct	Ch. 3	CS50 week 1
Functions: arguments, return values, purity, side effects	Ch. 4	Composing Programs 1.3–1.6
Recursion, base cases, the call stack	Ch. 5	SICP 1.2; Composing Programs 1.7
Collections and the cost of each operation	Ch. 6–7	Python's TimeComplexity page
Strings, encodings, and why UTF-8 exists	Ch. 8	Python's Unicode HOWTO
Files, streams, serialisation, JSON and CSV	Ch. 9	Python tutorial ch. 7
Errors and exceptions, failing loudly at the right boundary	Ch. 10	Python tutorial ch. 8
Automated testing from week one, with a real test runner	Ch. 11	pytest's getting-started guide
Debugging with a debugger	Ch. 12	CS50 week 2
Objects, classes, composition, interfaces, when not to inherit	Ch. 13–14	Python tutorial ch. 9; Composing Programs 2.5
Memory: the stack, the heap, references, mutation, aliasing	Ch. 5, 15	CS50 week 4; Python Tutor
Modules, packages, dependencies, virtual environments, lockfiles	Ch. 16	Python Packaging User Guide

Predict, run, explain. Each chapter's example is a complete Python program, written for Python 3.12 or later and shown with exactly what it printed. Before you read an output, cover it and predict what the program will print. Then run it, and when your prediction was wrong, find out why before going on: that gap is where the learning is. Every example prints the same thing on any machine. Random choices use a fixed seed, files go in temporary folders, and the network example runs its own small server. Where a chapter is about the cost of an operation, it counts the work, in copies, comparisons or probes, rather than timing it.

Do the stage's courses as well. This volume explains; the courses give you hundreds of hours of practice. Harvard's CS50 starts in C, and its week on memory and pointers makes chapter 15 concrete. Berkeley's CS 61A, whose free textbook is *Composing Programs*, or the book it grew from, *Structure and Interpretation of Computer Programs*, teaches abstraction and recursion more deeply than any other introduction. Exercism has about a hundred small problems per language with feedback from human mentors. Do one of CS 61A or SICP properly, with its exercises, rather than skimming both.

The deliverable is four programs, each a different shape: a command-line tool, a program that turns messy real-world data into a clean report, a small game with state, and a script that talks to a public API and caches what it gets. Chapters 18, 9, 19 and 20 build a small version of each. The appendix lists what the full versions need, and each must come with tests, a README, and no framework doing the thinking for you.

THE ONE HABIT

A program does exactly what it says, which is often not what you meant. Most of programming is closing the distance between the two, and you can only close what you can see.

So before you run code, predict what it will do, and when it surprises you, find out why before you change anything. A bug you understand is half fixed. A bug you made go away by changing things until it stopped is still there.

PART I

Thinking in code

Five chapters on the smallest pieces of a program and how they combine. Values and the expressions that compute them; names, and where a name can be seen; loops, and the invariants that make a loop correct instead of nearly correct; functions, the unit every larger program is built from; and recursion, which turns a problem into a smaller copy of itself. By the end of this part you can write a short program that does one thing, and explain exactly what happens when it runs.

Values, types and expressions

3 exercises

Everything a program works with is a value: a number, a piece of text, a list, the absence of a value. Every value has a type, and the type decides what operations mean, which is why `"42" * 2` is `"4242"` and `42 * 2` is 84. Programs compute new values from old ones with expressions, which Python evaluates one operation at a time in a fixed order. This chapter watches that happen step by step, and then looks at the places where Python's numbers behave differently from the numbers you learned at school.

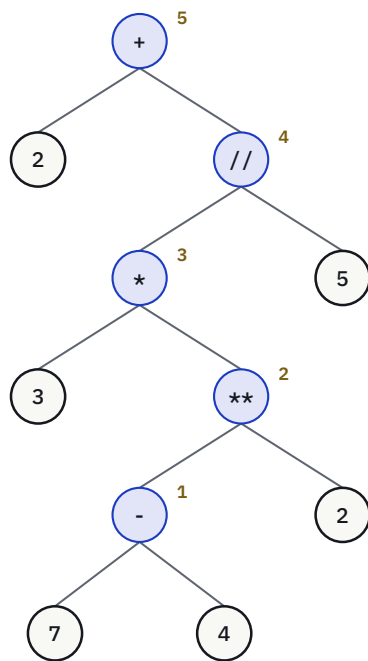
Values and their types

A **value** is a piece of data. `42` is a value, so is `"hello"`, so is `[4, 2]`. A **type** is a kind of value, together with the operations that make sense on it. Integers can be added and divided. Strings can be joined and searched. Lists can be appended to and sorted. Python's built-in types cover most of what a program needs: `int` for whole numbers, `float` for numbers with a fractional part, `str` for text, `bool` for `True` and `False`, `None` for "no value", and the collections `list`, `tuple`, `dict` and `set`, which hold other values and have chapters of their own.

`type(value)` tells you a value's type, and you'll use it constantly while learning. The type matters because operators mean different things for different types. `+` adds numbers but joins strings. `*` multiplies numbers but repeats strings and lists. And some combinations have no meaning at all: Python won't guess whether `"42" + 2` should be `"422"` or 44, so it raises a `TypeError` instead. That refusal is a feature. A language that guessed would sometimes guess wrong, silently.

Expressions

An **expression** is anything that produces a value: a literal like `7`, a name like `total`, a calculation like `2 + 3 * 4`, a function call like `len(items)`. Expressions combine: the operands of an operator can themselves be expressions, to any depth. Python evaluates a compound expression one operation at a time. It first decides the structure, which operations apply to which operands, using **precedence**, the rule that `*` binds more tightly than `+`. Then it computes operations whose operands are already values, replacing each with its result, until one value is left.



EVALUATED FROM THE BOTTOM UP

1	7 - 4	= 3	the parentheses come first
2	3 ** 2	= 9	** binds tighter than *
3	3 * 9	= 27	* and // share a level:
4	27 // 5	= 5	left to right
5	2 + 5	= 7	+ binds loosest, so it's last

Each step replaces one operation whose operands are already values with the value it produces.

The structure Python gives `2 + 3 * (7 - 4) ** 2 // 5`, and the order it evaluates the operations. Precedence decides the shape of the tree; evaluation then works from the leaves to the root.

The example's first part is a small program that performs those steps visibly. It uses Python's own parser, the `ast` module, to find the structure, then repeatedly evaluates the first operation whose operands are both values, and prints the expression after each step. The rest of the example looks at types, numbers and conversions.

EXAMPLE

values.py · an expression evaluated one step at a time, then types, integer and floating-point numbers, division, conversions and truthiness

```
import ast, math
from fractions import Fraction
from decimal import Decimal

# 1. An expression is evaluated one operation at a time. This stepper shows each step.
def steps(source):
    tree = ast.parse(source, mode="eval")
    def reduce(node):
        ready # evaluate the first operation that's
        if isinstance(node, ast.BinOp):
            if isinstance(node.left, ast.Constant) and isinstance(node.right, ast.Constant):
                return ast.Constant(eval(ast.unparse(node))), True
            node.left, done = reduce(node.left)
            if not done: node.right, done = reduce(node.right)
            return node, done
        if isinstance(node, ast.UnaryOp) and isinstance(node.operand, ast.Constant):
            return ast.Constant(eval(ast.unparse(node))), True
        return node, False
    out = [ast.unparse(tree)]
    while True:
        tree.body, done = reduce(tree.body)
```

```

        if not done: return out
        out.append(ast.unparse(tree))
for line in steps("2 + 3 * (7 - 4) ** 2 // 5"):
    print("    " + line)

# 2. Every value has a type, and the type decides what operations mean
for v in (42, 4.2, "42", True, None, [4, 2], (4, 2), {4: 2}):
    print(f"{v!r}>8} has type {type(v).__name__}")
print(f"\n'42' * 2 = {'42' * 2!r}, but 42 * 2 = {42 * 2}; '42' + 2 raises: ", end="")
try: "42" + 2
except TypeError as e: print(e)

# 3. Integers never overflow; floats are binary fractions and round
print(f"\n2 ** 100 = {2 ** 100:,}")
print(f"0.1 + 0.2 = {0.1 + 0.2!r}; equal to 0.3? {0.1 + 0.2 == 0.3}; "
      f"math.isclose: {math.isclose(0.1 + 0.2, 0.3)}")
print(f"0.1 is really {Decimal(0.1)}")
print(f"exact alternatives: Fraction(1, 10) * 3 = {Fraction(1, 10) * 3}, "
      f"Decimal('0.1') * 3 = {Decimal('0.1') * 3}")
print(f"1e308 * 10 = {1e308 * 10}; float('nan') == float('nan'): {float('nan') ==
      float('nan')}")

# 4. Three kinds of division, and rounding that surprises people
for a, b in ((7, 2), (-7, 2)):
    print(f"{a:>2} / {b} = {a / b:<5} {a:>2} // {b} = {a // b:<3} {a:>2} % {b} = {a % b}")
print(f"int(-3.7) = {int(-3.7)} (towards zero); math.floor(-3.7) = {math.floor(-3.7)}; "
      f"round(2.5) = {round(2.5)}, round(3.5) = {round(3.5)} (to even)")

# 5. Conversions and truthiness
print(f"\nint('42') = {int('42')}; bool('False') = {bool('False')}; bool('') = {bool('')}")
for text in ("4.2", "forty-two"):
    try: int(text)
    except ValueError as e: print(f"int({text!r}) raises ValueError: {e}")
falsy = [v for v in (0, 0.0, "", [], {}, None, "0", [0], " ") if not v]
print(f"falsy: {falsy}; everything else, including '0', [0] and ' ', is truthy")
print(f"and/or return an operand: 0 or 'default' = {0 or 'default'!r}; "
      f"'x' and 'y' = {'x' and 'y'!r}; [] and print('never') = {[] and print('never')!r}")

```

OUTPUT

```

2 + 3 * (7 - 4) ** 2 // 5
2 + 3 * 3 ** 2 // 5
2 + 3 * 9 // 5
2 + 27 // 5
2 + 5
7
    42 has type int
    4.2 has type float
    '42' has type str
    True has type bool
    None has type NoneType
    [4, 2] has type list
    (4, 2) has type tuple
    {4: 2} has type dict

'42' * 2 = '4242', but 42 * 2 = 84; '42' + 2 raises: can only concatenate str (not "int") to
str

2 ** 100 = 1,267,650,600,228,229,401,496,703,205,376
0.1 + 0.2 = 0.30000000000000004; equal to 0.3? False; math.isclose: True
0.1 is really 0.1000000000000000055511151231257827021181583404541015625
exact alternatives: Fraction(1, 10) * 3 = 3/10, Decimal('0.1') * 3 = 0.3
1e308 * 10 = inf; float('nan') == float('nan'): False
 7 / 2 = 3.5    7 // 2 = 3    7 % 2 = 1
-7 / 2 = -3.5  -7 // 2 = -4   -7 % 2 = 1
int(-3.7) = -3 (towards zero); math.floor(-3.7) = -4; round(2.5) = 2, round(3.5) = 4 (to even)

```

```
int('42') = 42; bool('False') = True; bool('') = False
int('4.2') raises ValueError: invalid literal for int() with base 10: '4.2'
int('forty-two') raises ValueError: invalid literal for int() with base 10: 'forty-two'
falsy: [0, 0.0, '', [], {}, None]; everything else, including '0', [0] and ' ', is truthy
and/or return an operand: 0 or 'default' = 'default'; 'x' and 'y' = 'y'; [] and print('never')
= []
```

The precedence rules you need to remember are few. From tightest to loosest: `**`; unary minus; `*`, `/`, `//` and `%`; `+` and `-`; comparisons such as `<` and `==`; `not`; `and`; `or`. Operators on the same level are evaluated left to right, except `**`, which goes right to left, so `2 ** 3 ** 2` is `2 ** 9`. When in doubt, add parentheses. They cost nothing, and they tell the next reader what you meant.

Integers and floating-point numbers

Python's integers have no upper limit: `2 ** 100` is computed exactly, using as much memory as the digits need. That's unusual. In C, Java and most other languages an integer is a fixed number of bits, typically 64, and arithmetic past the limit wraps around or is undefined. Python hides that, which is one less thing to go wrong.

Floats are different. A `float` is a **binary fraction** with about 16 significant decimal digits, stored in 64 bits under the IEEE 754 standard, which every modern processor implements in hardware. Most decimal fractions can't be written exactly in binary, just as $1/3$ can't be written exactly in decimal. So `0.1` is stored as the nearest binary fraction, which the example prints in full: slightly more than one tenth. Adding two such approximations gives `0.30000000000000004`, which isn't equal to the approximation of 0.3. This isn't a Python bug. Every language using hardware floats behaves the same way.

Three rules follow. Never compare floats with `==` after arithmetic; use `math.isclose(a, b)`, which allows a small relative error. Never use floats for money, where every cent must be right; use integers counting the smallest unit, cents or pence, or the `decimal` module, which does decimal arithmetic exactly. And remember that floats have limits: `1e308 * 10` is `inf`, and `nan`, "not a number", is the result of undefined operations like `inf - inf` and is famously not equal even to itself.

Division and rounding

Python has three division operators, and the example shows why it needs them. `/` is true division and always gives a float: `7 / 2` is 3.5. `//` is **floor division**: the result rounded down to a whole number, so `7 // 2` is 3 but `-7 // 2` is -4, because -4 is below -3.5. `%` is the remainder that goes with it, and in Python it takes the sign of the divisor, so `-7 % 2` is 1. Together they always satisfy `(a // b) * b + a % b == a`. C and Java truncate toward zero instead, so `-7 / 2` is -3 there and the remainder is -1. Code ported between languages breaks on exactly this.

Rounding has its own surprises. `int(-3.7)` truncates toward zero and gives `-3`, while `math.floor` gives `-4`. And `round(2.5)` is `2`, not `3`, because Python rounds halves to the nearest even number, a rule called **banker's rounding** that avoids a systematic upward bias when many values are rounded. If you need the schoolroom rule for money, the `decimal` module provides it.

Conversions and truthiness

Values change type only when you ask. `int("42")` parses a string into an integer and raises `ValueError` if the string isn't one. `str(42)` goes the other way. `float("2.5")` parses a float. Input from users, files and networks always arrives as text, so these conversions are where invalid data is first noticed, and chapter 10 is about what to do then.

Every value can also be used as a condition. A few values count as false: `False`, `None`, zero in any numeric type, and empty collections such as `"`, `[]` and `{}`. Everything else is true, including the string `"0"`, the list `[0]` and the string `"False"`. `and` and `or` use this **truthiness**, and they don't return `True` or `False`: they return one of their operands. `a or b` is `a` if `a` is true, otherwise `b`, and it doesn't evaluate `b` at all when it doesn't need to, which is called **short-circuiting**. That makes `name or "anonymous"` a common idiom, and also a common bug when a valid value, like `0`, happens to be false.

COMMON TRAPS

Comparing floats with `==` after any arithmetic. Use `math.isclose`.

Using floats for money. Count cents in an `int`, or use `decimal.Decimal`.

Assuming `//` and `%` work as in C or Java for negative numbers.

Writing `x or default` when `0`, an empty string or an empty list is a legitimate value of `x`.

Expecting `round(2.5)` to be `3`. It rounds halves to even.

EXERCISES

EXERCISE 1.1 Predict, then check

Without running them, write down the values of `2 ** 3 ** 2`, `-2 ** 2`, `7 // -2`, `7 % -2` and `10 / 4 * 2`. Then check them in Python and explain any you got wrong.

SOLUTION

`2 ** 3 ** 2` is 512: `**` groups from the right, so it's `2 ** 9`, not `8 ** 2`. `-2 ** 2` is `-4`, because `**` binds more tightly than unary minus, so it means `-(2 ** 2)`; write `(-2) ** 2` for 4. `7 // -2` is `-4`: the exact quotient is `-3.5`, and floor division rounds down, toward minus infinity. `7 % -2` is `-1`, since the remainder takes the divisor's sign, and it checks out: $(-4) \times (-2) + (-1) = 7$. `10 / 4 * 2` is `5.0`: `/` and `*` share a level and go left to right, so it's `(10 / 4) * 2`, and `/` always produces a float. If you got the second one wrong, you're in good company, and it's a case where parentheses are worth writing even though they aren't needed.

EXERCISE 1.2 Ten dimes

Add `0.1` to a total ten times, starting from 0, and compare the result with 1.0. What do you get, and why? Rewrite the calculation two ways so that it's exact.

SOLUTION

The total is `0.9999999999999999`, and `total == 1.0` is `False`. Each `0.1` is stored as a binary fraction slightly larger than one tenth, and each addition rounds its result to the nearest representable float; the ten small errors don't cancel exactly. The first exact version counts in whole cents: add the integer 10 ten times, get 100, and divide by 100 only when displaying. Integers are exact, so there's nothing to go wrong. The second uses decimals: `from decimal import Decimal`, then add `Decimal("0.1")` ten times to get exactly `Decimal("1.0")`. Note the string: `Decimal(0.1)`, from a float, would faithfully copy the float's error, as the example shows. Financial code uses one of these two approaches, never floats.

EXERCISE 1.3 When zero is an answer

A function receives `quantity`, which is a number or `None` when the user left the field blank. It does `quantity = quantity or 1` to default blanks to 1. What goes wrong, and what's the correct version?

SOLUTION

A user who enters 0 gets 1. `or` tests truthiness, and 0 is false, so `0 or 1` evaluates to 1, exactly as `None or 1` does. The code can't tell "no answer" from "the answer zero". The fix tests for the specific value meaning "missing": `quantity = 1 if quantity is None else quantity`. `is None` is the right test here, because there's only one `None` object, and it's what Python's style guide recommends for this

comparison. The same bug appears with empty strings and empty lists, for instance `name = name or "anonymous"` silently replacing a name the user deliberately cleared. Use truthiness only when every false value really means the same thing.

CHECKLIST

- ☐ Can you name Python's basic types and find any value's type with `type`?
- ☐ Can you evaluate a compound expression by hand, in the order Python does?
- ☐ Can you explain why `0.1 + 0.2 != 0.3`, and how to handle money correctly?
- ☐ Can you predict `/`, `//` and `%` for negative operands?
- ☐ Can you list the falsy values, and explain what `a or b` returns?

GO DEEPER

- DOCS** The Python Tutorial: An Informal Introduction to Python Python documentation
- DOCS** Floating-Point Arithmetic: Issues and Limitations Python documentation
- BOOK** Composing Programs, 1.2: Elements of Programming John DeNero

Names, assignment and scope

3 exercises

A variable in Python is not a box that holds a value. It's a name attached to a value, and two names can be attached to the same one. That single fact explains most of the surprises beginners meet, from a list that changes "by itself" to a counter that refuses to count. This chapter shows how assignment binds names, how mutating a value differs from rebinding a name, and where Python looks when you use a name: the rules called scope.

Assignment binds a name

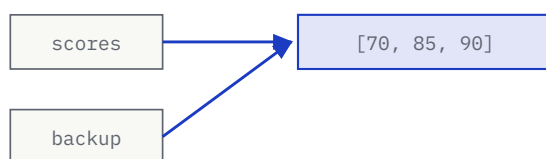
A **statement** is an instruction that does something rather than producing a value, and the most common is assignment. `scores = [70, 85]` evaluates the right side, creating a list, and then binds the name `scores` to it. `backup = scores` evaluates the right side again, which is just the name `scores`, so it finds the same list, and binds a second name to it. Nothing was copied. There is one list with two names, and appending through either name changes the one list, which is visible through both.

Now assign `backup = [0]`. That creates a new list and moves the name `backup` to it. The original list is untouched, and `scores` still names it. So there are two quite different operations that both look like "changing a variable": **mutating** an object, through a method like `append` or an assignment to an element like `xs[0] = 1`, which every name for that object sees; and **rebinding** a name with `=`, which affects that name only.

```
BACKUP = SCORES; BACKUP.APPEND(90)
```

NAMES

OBJECTS

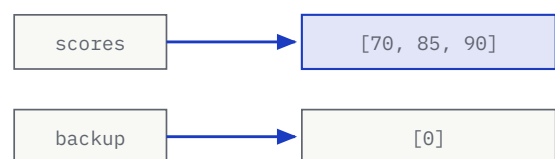


two names, one list: a change through either is seen by both

```
THEN BACKUP = [0]
```

NAMES

OBJECTS



rebinding moves a name; the list itself is untouched

Names point to objects. Assignment moves a name; mutation changes an object, and every name pointing at it sees the change.

Some types can't be mutated at all: integers, floats, strings, tuples and `None` are **immutable**. For them, an operation that looks like a change, such as `count += 1`, computes a new value and rebinds the name to it. That's why `before = count; count += 1` leaves `before` alone. For a mutable list, `+=` really does change the object in place, so it's seen

through every other name, as the example shows. The same symbol does two different things depending on the type. Chapter 15 returns to this in detail, because it's the root of the aliasing bugs that experienced programmers still write.

Names have no types; values do

A name can be bound to an integer on one line and a string on the next. Python checks types when an operation runs, not when a name is assigned, so a name's "type" is just the type of whatever it's bound to at the moment. Type hints such as `count: int = 0` document your intent and let checking tools catch mistakes, as chapter 6 of *The Engineer's Press* explains, but Python itself ignores them when running the program. Good names do more for correctness than you might expect: `total_pence` tells the reader the unit and the type, and `x2` tells them nothing. Python's conventions, from the style guide PEP 8, are `lower_case_with_underscores` for variables and functions, `CapitalisedWords` for classes and `UPPER_CASE` for constants.

EXAMPLE

names.py · binding, mutation and rebinding; the LEGB lookup order; shadowing a built-in; `UnboundLocalError`, global and nonlocal; and which blocks make scopes

```
import builtins

# 1. Assignment binds a name to an object. Two names can be bound to one object.
scores = [70, 85]
backup = scores                # not a copy: a second name for the same list
backup.append(90)              # mutate the object through one name ...
print(f"scores = {scores}, backup is scores: {backup is scores}")    # ... and both names see it
backup = [0]                   # rebinding a name leaves the object alone
print(f"after rebinding backup: scores = {scores}, backup = {backup}")

count = 5
before = count
count += 1                     # ints can't change, so += makes a new int and rebinds count
print(f"count = {count}, before = {before}")
total = scores
total += [100]                 # but on a list, += changes the list in place
print(f"after total += [100], scores = {scores}")

a, b = 1, 2
a, b = b, a                    # the right side is built first, then unpacked: a swap
print(f"swapped: a = {a}, b = {b}")

# 2. Where Python looks for a name: Local, Enclosing function, Global, then Built-in
level = "global"
def outer():
    level = "enclosing"
    def inner():
        found = "built-in" if len is builtins.len else "shadowed"
        return f"inner sees {level!r}; len is the {found} one"
    return inner()
print("\n" + outer())

list = ["a", "b"]              # shadowing a built-in with a global name
try: list("xyz")
except TypeError as e: print(f"after list = [...], list('xyz') fails: {e}")
```

```

del list                                # the global goes; the built-in is visible again
print(f"after del list: list('xyz') = {list('xyz')}")

# 3. Assigning anywhere in a function makes the name local to the whole function
visits = 0
def record_visit():
    visits += 1                          # reads the local visits, which has no value yet
    try: record_visit()
    except UnboundLocalError as e: print(f"\nrecord_visit(): UnboundLocalError: {e}")
def record_visit_fixed():
    global visits                        # say so explicitly, and the name means the global
    visits += 1
record_visit_fixed(); record_visit_fixed()
print(f"with global: visits = {visits}")

def make_counter():
    n = 0
    def step():
        nonlocal n                      # the same rule one level out: n belongs to make_counter
        n += 1
        return n
    return step
step = make_counter()
print(f"a counter with nonlocal: {[step() for _ in range(3)]}")

# 4. Blocks don't make scopes: a loop variable outlives its loop; a comprehension's doesn't
for i in range(3):
    pass
print(f"\nafter the for loop, i = {i}")
squares = [j * j for j in range(3)]
print(f"after the comprehension, j defined: {'j' in globals()}")
if (n := len(squares)) > 2:               # := assigns inside an expression
    print(f"the walrus bound n = {n} inside an if condition")

```

OUTPUT

```

scores = [70, 85, 90], backup is scores: True
after rebinding backup: scores = [70, 85, 90], backup = [0]
count = 6, before = 5
after total += [100], scores = [70, 85, 90, 100]
swapped: a = 2, b = 1

inner sees 'enclosing'; len is the built-in one
after list = [...], list('xyz') fails: 'list' object is not callable
after del list: list('xyz') = ['x', 'y', 'z']

record_visit(): UnboundLocalError: cannot access local variable 'visits' where it is not
    associated with a value
with global: visits = 2
a counter with nonlocal: [1, 2, 3]

after the for loop, i = 2
after the comprehension, j defined: False
the walrus bound n = 3 inside an if condition

```

Scope: where Python looks for a name

A name's **scope** is the part of the program where it can be used. When Python meets a name, it looks in four places, in order, and uses the first match. That's the **LEGB** rule: the Local scope of the current function; the scopes of any Enclosing functions it's written

inside; the **Global** scope of the module; and the **Built-in** scope, where `len`, `print` and `list` live. In the example, `inner` has no `level` of its own, so it finds the enclosing function's, which hides the global one.

The built-ins come last, so any name you define hides one. `list = ["a", "b"]` is legal, and afterwards `list("xyz")` fails, because `list` now means your list. This is called **shadowing**, and it's a common source of baffling errors when someone names a variable `list`, `dict`, `str`, `sum`, `max` or `input`. Linters such as Ruff warn about it.

The rule that surprises everyone is how Python decides a name is local. It doesn't look at the order of lines. If a function assigns to a name anywhere in its body, that name is local everywhere in the function, decided before the function runs. So `visits += 1`, which reads `visits` and then assigns it, makes `visits` local, and the read happens before any local value exists: `UnboundLocalError`. Declaring `global visits` tells Python the name means the module's variable, and `nonlocal n` does the same for an enclosing function's variable, which is how the example's counter keeps its count between calls.

Use those declarations sparingly. A function that changes a global is harder to test and reason about, because its behaviour depends on state outside it. Usually the better design is to pass the value in and return the new value, which chapter 4 develops. The counter built with `nonlocal` is a legitimate use, because the state is private to the function that owns it, and chapter 13 shows the same idea done with a class.

Blocks are not scopes

In many languages, a variable declared inside an `if` or a loop disappears at the end of it. In Python, only functions, classes, modules and comprehensions create scopes. A `for` loop's variable is an ordinary name in the surrounding scope and keeps its last value after the loop ends; a name first assigned inside an `if` exists afterwards, if that branch ran. A list comprehension is the exception: its variable lives in the comprehension's own scope and doesn't leak, as the example checks. Finally, the walrus operator `:=`, added in Python 3.8, is assignment inside an expression, useful in conditions such as `if (n := len(items)) > 10:`. Ordinary assignment with `=` is a statement and can't appear there.

COMMON TRAPS

Thinking `b = a` copies a list. It names the same list; use `a.copy()` or `list(a)` for a copy.

Naming a variable `list`, `str`, `sum` or `input`, and then calling the built-in.

Incrementing a global inside a function without `global`, and getting `UnboundLocalError`.

Reaching for `global` when returning a value would be clearer.

Using a loop variable after the loop without noticing that it keeps its last value, or doesn't exist

if the loop never ran.

EXERCISES

EXERCISE 2.1 Plus or plus-equals

After `a = [1]; b = a; a = a + [2]`, what are `a` and `b`? After `a = [1]; b = a; a += [2]`? Draw the names and objects for each.

SOLUTION

In the first, `a` is `[1, 2]` and `b` is `[1]`. `a + [2]` builds a new list, and `a = ...` moves the name `a` to it; `b` still names the original. Two names, two lists. In the second, both are `[1, 2]`. For lists, `+=` calls the list's in-place extend method, which changes the existing object, and then rebinds `a` to that same object. One list, two names. The difference only matters when another name refers to the list, which is exactly when it causes bugs: a function that does `items += [x]` on a list it was given changes the caller's list, while `items = items + [x]` doesn't.

EXERCISE 2.2 A counter without global

Rewrite `record_visit` from the example so that it needs neither `global` nor `nonlocal`, and so that a test could check it without touching module state.

SOLUTION

Make the count an input and the new count an output: `def record_visit(visits): return visits + 1`, called as `visits = record_visit(visits)`. The function now depends only on its argument, so a test is a single line, `assert record_visit(4) == 5`, and two parts of the program can keep separate counts without interfering. The caller decides where the state lives. If the state belongs with other related data, such as a visit count and the time of the last visit, the next step is a small class with a `record_visit` method, which chapter 13 covers. What's best avoided is a function that reaches out and changes a variable somewhere else, invisible from the call site.

EXERCISE 2.3 The vanishing built-in

A script contains `sum = 0` near the top, and a hundred lines later `average = sum(scores) / len(scores)` fails with `TypeError: 'int' object is not callable`. Explain the error, and describe two ways a tool could have caught it before the script ran.

SOLUTION

`sum = 0` created a global named `sum`, which comes before the built-ins in the LEGB order. From then on, `sum` means the integer 0, so `sum(scores)` tries to call 0 as a function. The message says exactly that, though it doesn't say why `sum` is an integer. The fix is to rename the variable, to `total` for instance. A linter catches it statically: Ruff's rule A001 and flake8's builtins plugin report any assignment that shadows a built-in. A type checker such as mypy or Pyright catches it too, by inferring that `sum` is an `int` and reporting that an `int` isn't callable. Both run in seconds and are worth running on every save.

CHECKLIST

- ☐ Can you explain the difference between rebinding a name and mutating an object?
- ☐ Can you predict when `+=` changes an object in place and when it makes a new one?
- ☐ Can you state the LEGB rule and say which scope a name in a nested function comes from?
- ☐ Can you explain `UnboundLocalError` and fix it without a global?
- ☐ Can you say which constructs create a scope in Python and which don't?

GO DEEPER

ARTICLE Facts and myths about Python names and values Ned Batchelder

DOCS The Python Language Reference: Naming and binding Python documentation

BOOK Composing Programs, 1.3: Defining New Functions, with environment diagrams John DeNero

Control flow and loop invariants

3 exercises

*A program becomes interesting when it decides and repeats: `if` chooses between paths, and loops do something once per item or until a condition changes. Loops are also where most off-by-one errors live, because a loop that is nearly right looks exactly like one that is right. The remedy is to know what your loop keeps true on every pass, its **invariant**, and why it must stop. This chapter writes two classic loops with their invariants checked, then tests three versions of binary search against every small input and finds the two bugs at once.*

Choosing and repeating

`if`, `elif` and `else` run the first branch whose condition is true. Python has two loops. A `for` loop runs once for each item of something iterable: a list, a string, a file's lines, or `range(n)`, which produces 0 up to `n - 1`. A `while` loop runs as long as its condition is true, and is for loops whose length you don't know in advance. Inside either, `break` leaves the loop at once and `continue` skips to the next pass. A `for` loop can also have an `else` clause, which runs only if the loop finished without a `break`; it's the natural shape for "search, and do something if nothing was found", as the end of the example shows.

Prefer `for` when you can. It can't run forever over a finite collection, and it can't skip or repeat an item through a mistake with an index. `enumerate(items)` gives you the index too, and `zip(a, b)` walks two sequences together. A `while` loop puts the burden of progress on you: something in the body must eventually make the condition false.

Invariants

A **loop invariant** is a statement about your variables that is true before the loop starts and stays true after every pass. It's the loop's purpose, stated so it can be checked. To show a loop is correct you show three things. **Initialisation**: the invariant is true before the first pass. **Maintenance**: if it's true before a pass, the pass keeps it true. **Termination**: when the loop stops, the invariant together with the reason it stopped gives the result you wanted.

Insertion sort is the standard example. Its invariant is "`items[:i]` is sorted". It starts true, because a list of one item is sorted. Each pass takes `items[i]`, shifts the larger sorted items right by one and drops it into the gap, so `items[:i + 1]` is sorted: maintained. The loop stops when `i` reaches the length, so the whole list is sorted. The example asserts the invariant on every pass and prints the list as it grows, so you can watch it hold.

EXAMPLE

loops.py · insertion sort with its invariant asserted, binary search and two buggy versions tested against every small case, and for-else

```
# 1. Insertion sort, checking its loop invariant on every pass: items[:i] is always sorted
def insertion_sort(items, show=False):
    items = list(items)
    for i in range(1, len(items)):
        assert items[:i] == sorted(items[:i]), "invariant broken"      # true before each pass
        current, j = items[i], i - 1
        while j >= 0 and items[j] > current:      # shift bigger items right to make a gap
            items[j + 1] = items[j]
            j -= 1
        items[j + 1] = current
        if show: print(f"    after pass {i}: sorted {items[:i + 1]} | still to do {items[i + 1:]}")
    return items
insertion_sort([5, 2, 9, 1, 7], show=True)

# 2. Binary search. Invariant: if target is in xs, it's in xs[lo:hi]. Variant: hi - lo shrinks.
def search(xs, target):
    lo, hi, steps = 0, len(xs), 0
    while lo < hi:
        steps += 1
        mid = (lo + hi) // 2
        if xs[mid] < target: lo = mid + 1      # target can't be at mid or left of it
        elif xs[mid] > target: hi = mid      # target can't be at mid or right of it
        else: return mid, steps
    return None, steps

def search_bug1(xs, target):      # an off-by-one: the last element is never
    checked                       checked
    lo, hi = 0, len(xs) - 1
    while lo < hi:
        mid = (lo + hi) // 2
        if xs[mid] < target: lo = mid + 1
        elif xs[mid] > target: hi = mid
        else: return mid
    return None

def search_bug2(xs, target, limit=100):      # lo = mid instead of mid + 1: it can stop
    shrinking
    lo, hi, n = 0, len(xs), 0
    while lo < hi:
        n += 1
        if n > limit: return "stuck"
        mid = (lo + hi) // 2
        if xs[mid] < target: lo = mid
        elif xs[mid] > target: hi = mid
        else: return mid
    return None

def check(fn):
    """Every sorted list of distinct values from 0, 2, 4, ... of length 0 to 6, every
    target."""
    cases = failures = 0; first = None
    for n in range(7):
        xs = list(range(0, 2 * n, 2))
        for target in range(-1, 2 * n + 1):
            got = fn(xs, target)
            got = got[0] if isinstance(got, tuple) else got
            want = xs.index(target) if target in xs else None
            cases += 1
            if got != want:
                failures += 1; first = first or (xs, target, want, got)
    return cases, failures, first
```

```

print()
for name, fn in (("correct", search), ("bug 1: hi = len - 1", search_bug1),
                 ("bug 2: lo = mid", search_bug2)):

    cases, failures, first = check(fn)
    detail = (f"; first: search({first[0]}, {first[1]}) gave {first[3]!r}, wanted {first[2]!r}"
              if first else "")
    print(f"{name:20} {cases} cases, {failures} wrong{detail}")
big = list(range(1_000_000))
worst = max(search(big, t)[1] for t in (0, 499_999, 999_999, -5))
print(f"\nsearching a million items takes at most {worst} steps")

# 3. for ... else: the else runs only if the loop wasn't left by break
def first_negative(nums):
    for i, x in enumerate(nums):
        if x < 0:
            break
    else:
        return "none"
    return f"index {i}"
print(f"first negative in [3, 1, -4, 1]: {first_negative([3, 1, -4, 1])}; "
      f"in [3, 1]: {first_negative([3, 1])}")

```

OUTPUT

```

after pass 1: sorted [2, 5] | still to do [9, 1, 7]
after pass 2: sorted [2, 5, 9] | still to do [1, 7]
after pass 3: sorted [1, 2, 5, 9] | still to do [7]
after pass 4: sorted [1, 2, 5, 7, 9] | still to do []

correct                56 cases, 0 wrong
bug 1: hi = len - 1    56 cases, 6 wrong; first: search([0], 0) gave None, wanted 0
bug 2: lo = mid        56 cases, 27 wrong; first: search([0], 1) gave 'stuck', wanted None

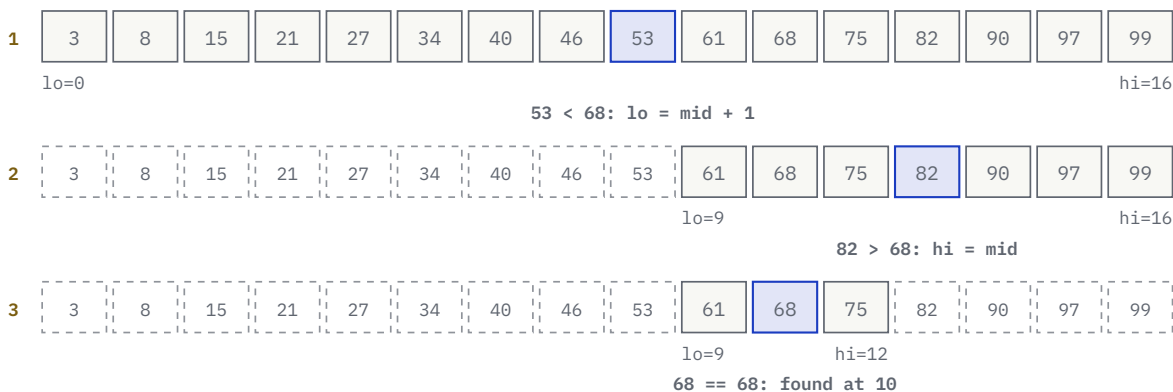
searching a million items takes at most 20 steps
first negative in [3, 1, -4, 1]: index 2; in [3, 1]: none

```

Binary search, and how nearly right goes wrong

Binary search finds a value in a sorted list by repeatedly halving the range that could contain it. It is famous for being easy to describe and hard to write: Jon Bentley reported that most professional programmers he asked couldn't write a correct version, and a bug in the version in Java's standard library went unnoticed for nine years. The invariant makes it manageable. Use a half-open range, `lo` included and `hi` excluded, as Python's slices and `range` do, and keep this true: **if the target is in the list, it's in `xs[lo:hi]`**.

SEARCHING FOR 68: THE TARGET, IF PRESENT, IS ALWAYS IN `xs[lo:hi]`



dashed: ruled out. Each step halves what's left, so 16 items need at most 5 looks.

The invariant at work. Everything outside `xs[lo:hi]` has been ruled out, and every comparison rules out at least half of what's left.

Each branch of the correct version preserves the invariant. If `xs[mid] < target`, the target can't be at `mid` or anywhere left of it, because the list is sorted, so `lo = mid + 1`. If `xs[mid] > target`, it can't be at `mid` or to its right, so `hi = mid`, which excludes `mid` because `hi` is excluded. When `lo == hi`, the range is empty, and the invariant says the target isn't present. Termination needs one more argument, a **variant**: a whole number that decreases on every pass and can't go below zero. Here it's `hi - lo`, and both branches shrink it by at least one, because `mid` is always inside the range.

The two buggy versions each break one of those arguments. Bug 1 starts with `hi = len(xs) - 1` but keeps the half-open logic, so the last element is never inside the range: it fails exactly once per list length, 6 times in 56 cases, always on the last element. Bug 2 writes `lo = mid`, which doesn't shrink the range when `mid == lo`, so the variant stops decreasing and the loop spins forever; the example caps it and reports "stuck". Neither bug shows up on a typical hand-picked test. Testing every list of length 0 to 6 against every possible target, 56 cases, found both at once, which is a technique chapter 11 develops: small inputs, all of them.

The payoff for getting it right is large. Each look halves the range, so a million items take at most 20 looks, as the example measures, and a billion take 30. A loop that scanned item by item would take up to a million and a billion.

COMMON TRAPS

Mixing inclusive and exclusive bounds in one loop. Pick half-open ranges, as Python does, and stick to them.

A **while** loop whose body doesn't always make progress, so some input makes it run forever.

Using indexes where `for x in items`, `enumerate` or `zip` would do.

Testing only the middle of the input space, when bugs live at the empty list, one item, and the last item.

Forgetting that `else` on a loop means "no break", not "the loop didn't run".

EXERCISES

EXERCISE 3.1 The invariant of max

Write a loop that finds the largest number in a non-empty list without using `max`. State its invariant, and check initialisation, maintenance and termination.

SOLUTION

```
def largest(xs):
    best = xs[0]
    for i in range(1, len(xs)):
        # invariant: best == max(xs[:i])
        if xs[i] > best: best = xs[i]
    return best
```

The invariant is "`best` is the largest of `xs[:i]`". Initialisation: before the first pass, `i` is 1 and `best` is `xs[0]`, the largest of the one-element slice. Maintenance: the pass compares `xs[i]` with the largest so far and keeps whichever is bigger, so afterwards `best` is the largest of `xs[:i + 1]`. Termination: the loop ends after `i` reaches `len(xs) - 1`, so `best` is the largest of the whole list. The precondition "non-empty" matters: on an empty list, `xs[0]` raises `IndexError`, which is a better outcome than returning a made-up value, and the docstring should say so.

EXERCISE 3.2 Why bug 2 is stuck

Trace `search_bug2([0], 1)` from the example by hand. Which part of the correctness argument fails, and is the invariant itself still true?

SOLUTION

Start with `lo = 0`, `hi = 1`. `mid = (0 + 1) // 2 = 0`, and `xs[0] = 0 < 1`, so `lo = mid = 0`. Nothing changed, so the next pass computes the same `mid` and does the same thing, forever. The invariant still holds: 1 isn't in the list, so "if it's in the list, it's in `xs[0:1]`" is vacuously

true, and in general `lo = mid` never excludes a position that could hold the target. What fails is termination. The variant `hi - lo` must decrease on every pass, and when `mid == lo`, which happens whenever the range has one element, `lo = mid` leaves it at 1. That's why correct code writes `mid + 1 : mid` was just checked and found too small, so excluding it is both allowed by the invariant and needed for progress.

EXERCISE 3.3 Counting digits

Write `digits(n)`, which returns the number of decimal digits in a non-negative integer, using a `while` loop and no strings. Give the variant that proves it stops, and test it on every value from 0 to 1,000 against `len(str(n))`.

SOLUTION

```
def digits(n):
    count = 1
    while n >= 10:
        n //= 10
        count += 1
    return count

assert all(digits(n) == len(str(n)) for n in range(1001))
```

The variant is `n` itself: each pass replaces it with `n // 10`, which is strictly smaller for any `n ≥ 10`, and a non-negative whole number can't decrease forever. Starting `count` at 1 handles 0 correctly, which has one digit; a version that starts at 0 and loops `while n > 0` returns 0 for 0, a bug that only the edge case reveals. The exhaustive check takes a moment and covers every one-, two-, three- and four-digit boundary, which is far more convincing than three hand-picked tests.

CHECKLIST

- ☐ Can you choose between `for` and `while`, and use `enumerate`, `zip` and `for-else`?
- ☐ Can you state a loop's invariant and check initialisation, maintenance and termination?
- ☐ Can you give a variant that proves a `while` loop stops?
- ☐ Can you write binary search with half-open bounds, correct first time?

☐ Can you test a loop against every small input instead of a few chosen ones?

GO DEEPER

DOCS The Python Tutorial: More Control Flow Tools [Python documentation](#)

NOTE Why numbering should start at zero (EWD 831) [Edsger W. Dijkstra](#)

ARTICLE Nearly All Binary Searches and Mergesorts are Broken [Joshua Bloch](#)

Functions

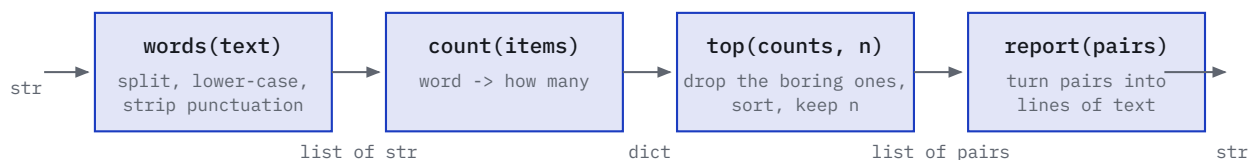
3 exercises

A function gives a name to a computation, so you can use it without thinking about how it works. That's the most important tool in programming, because it's how a problem too big to hold in your head becomes several problems that each fit. This chapter breaks a small text-analysis problem into four functions, then covers what makes a function easy to use and test: returning results instead of changing things, the trap of mutable default arguments, the ways of passing arguments, and functions as values in their own right.

Breaking a problem into functions

"Find the most common interesting words in a passage" is one sentence, but the program has several steps: split the text into words, ignoring case and punctuation; count each word; drop the boring words and take the most frequent; format the result. Each step is a **function**: a named piece of code that takes **arguments** and **returns** a result. `def words(text):` declares a function with one **parameter**, `text`, which is bound to whatever argument the caller passes, and `return` sends a value back. A function that reaches the end without `return` returns `None`.

Each function takes one value and returns one value; none of them prints or changes its input.



`report(top(count(words(TEXT)), 5, ignore=...))` is the whole program, and each piece can be tested alone.

The example's program as a pipeline of functions. Each has one job, and the type of value passed between them is the contract between neighbours.

Decomposed like this, each function is small enough to get right by itself, and to test by itself, because its whole behaviour is "this input gives that output". When the program is wrong, you can check each stage's output separately and find the stage where it goes wrong. And the pieces are reusable: `count` works for anything you can iterate over, not just words. A useful test for whether a function is well chosen: can you say what it does in one sentence without the word "and"?

EXAMPLE

functions.py · a problem split into small functions, pure and impure calls, the mutable default trap, argument kinds, and functions as values

```
import string

TEXT = """It was the best of times, it was the worst of times, it was the age of wisdom,
it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity."""

# 1. A problem broken into small functions, each with one job, input in and result out
def words(text):
    """Lower-case words, with punctuation stripped."""
    return [w.strip(string.punctuation) for w in text.lower().split()]
def count(items):
    counts = {}
    for item in items:
        counts[item] = counts.get(item, 0) + 1
    return counts
def top(counts, n, ignore=()):
    kept = [(word, c) for word, c in counts.items() if word not in ignore]
    return sorted(kept, key=lambda pair: (-pair[1], pair[0]))[:n] # most common, then a-z
def report(pairs):
    return "\n".join(f"    {word:<8} {'#' * c} {c}" for word, c in pairs)
print(report(top(count(words(TEXT)), 5, ignore={"the", "of", "it", "was"})))

# 2. Pure functions return a result; impure ones change something. Know which you're calling.
nums = [3, 1, 2]
result = sorted(nums) # pure: returns a new list, leaves nums alone
print(f"\nsorted(nums) = {result}, nums still {nums}")
result = nums.sort() # impure: sorts nums in place, and returns None
print(f"nums.sort() returned {result}; nums is now {nums}")

# 3. A default value is evaluated once, when the def runs. A mutable default is shared.
def add_tag(tag, tags=[]):
    tags.append(tag)
    return tags
print(f"\nadd_tag('a') = {add_tag('a')}, then add_tag('b') = {add_tag('b')} (the same list!)")
def add_tag_fixed(tag, tags=None):
    tags = [] if tags is None else tags
    tags.append(tag)
    return tags
print(f"fixed: {add_tag_fixed('a')}, then {add_tag_fixed('b')}")

# 4. The ways to pass arguments
def describe(name, /, age, *hobbies, city="nowhere", **extra):
    return f"{name}, {age}, hobbies {hobbies}, city {city}, extra {extra}"
print("\n" + describe("Ada", 36, "maths", "poetry", city="London", title="Countess"))
for bad in ("describe(name='Ada', age=36)", "describe('Ada')"):
    try: eval(bad)
    except TypeError as e: print(f"{bad} -> TypeError: {e}")

# 5. Functions are values: store them, pass them, return them
def make_scaler(factor):
    def scale(x): return x * factor
    return scale
double, triple = make_scaler(2), make_scaler(3)
ops = {"double": double, "triple": triple, "negate": lambda x: -x}
print(f"\n{(name, op(7)) for name, op in ops.items()}")
print(f"sorted by length, then a-z: {sorted(['pear', 'fig', 'banana', 'kiwi'], key=lambda w:
    (len(w), w))}")
```

OUTPUT

```

age      ## 2
epoch    ## 2
times    ## 2
belief   # 1
best     # 1

sorted(nums) = [1, 2, 3], nums still [3, 1, 2]
nums.sort() returned None; nums is now [1, 2, 3]

add_tag('a') = ['a'], then add_tag('b') = ['a', 'b'] (the same list!)
fixed: ['a'], then ['b']

Ada, 36, hobbies ('maths', 'poetry'), city London, extra {'title': 'Countess'}
describe(name='Ada', age=36) -> TypeError: describe() missing 1 required positional argument:
    'name'
describe('Ada') -> TypeError: describe() missing 1 required positional argument: 'age'

[('double', 14), ('triple', 21), ('negate', -7)]
sorted by length, then a-z: ['fig', 'kiwi', 'pear', 'banana']

```

Pure functions and side effects

A **pure function's** result depends only on its arguments, and calling it changes nothing else: no printing, no modifying its arguments, no writing files or globals. The four pipeline functions are pure. Anything else a function does is a **side effect**. Programs need side effects, since a program that prints nothing and writes nothing is useless, but they're easiest to understand and test when they're kept at the edges, in a few functions whose job is input and output, with the logic in pure functions between them.

Python's standard library follows a convention that makes the distinction visible: a method that changes an object in place returns `None`. `sorted(nums)` is pure and returns a new list; `nums.sort()` sorts the list itself and returns `None`, so `result = nums.sort()` is a classic bug that leaves `result` as `None`. The same holds for `list.append`, `list.reverse` and `dict.update`. Following the same convention in your own code, a function either answers a question or makes a change, but not both, makes calls easier to read.

The mutable default argument

A **default value** makes a parameter optional: `def add_tag(tag, tags=[])`. The surprise is when the default is evaluated: once, when the `def` statement runs, not each time the function is called. So every call that uses the default gets the same list, and the example's second call returns `['a', 'b']`, because the first call's tag is still in the shared list. Immutable defaults such as numbers, strings and `None` are safe, since they can't be changed. For a mutable default, the idiom is `tags=None`, then `if tags is None: tags = []` inside the function, which makes a fresh list on every call. Linters flag the mistake, and it's worth letting them.

Passing arguments

Arguments can be passed by position, `describe("Ada", 36)`, or by name, `describe("Ada", age=36)`. Named arguments make calls readable when there are several of the same type: `resize(img, width=640, height=480)` can't be misread the way `resize(img, 640, 480)` can. A function can also accept any number of extra positional arguments with `*args`, collected as a tuple, and extra keyword arguments with `**kwargs`, collected as a dict. In the signature, a bare `/` makes the parameters before it positional-only, and a bare `*`, or `*args`, makes the ones after it keyword-only. The example's `describe` uses all of them, and the error messages for bad calls show Python checking the call against the signature before running a line of the body.

Keyword-only parameters are worth using deliberately. `def connect(host, *, timeout=10, retries=3)` forces callers to write `timeout=`, so a later reader never has to guess what `connect("db", 5, 2)` means, and you can add or reorder options later without breaking callers.

Functions are values

A function is an object like any other: it can be stored in a variable, kept in a dict, passed to another function and returned from one. `sorted(words, key=len)` passes the function `len`, and `sorted` calls it on each item to decide the order. A `lambda` is a small anonymous function written inline, `lambda w: (len(w), w)`, for when a name would be overkill. And a function defined inside another can use the outer function's variables even after the outer one has returned, which is how `make_scaler(2)` returns a function that remembers its factor. That's a **closure**, and chapter 10 of *The Engineer's Press* explains how it's implemented.

Functions as values let you write code that's general over behaviour, not just data. A dict of functions replaces a long `if - elif` chain that dispatches on a name. A sort or search takes the comparison as an argument. Chapter 14 returns to this: often the right way to vary one behaviour is to pass a function, not to write a subclass.

COMMON TRAPS

Using a mutable default argument such as `[]` or `{}`, shared between calls.

Assigning the result of an in-place method, as in `xs = xs.sort()`, and getting `None`.

Functions that both compute and print, which can't be reused or tested without capturing output.

Long positional argument lists where a swapped pair of numbers goes unnoticed. Use keywords.

Writing one long function because the parts "aren't reused". Small functions are for reading and testing, not only for reuse.

EXERCISE 4.1 Separate the printing

This function computes a receipt and prints it. Split it into a pure function and a printing function, and write one test for the pure part.

```
def receipt(prices):
    total = 0
    for name, p in prices.items():
        print(f"{name:10} {p:6.2f}")
        total += p
    print(f"{'TOTAL':10} {total:6.2f}")
```

SOLUTION

```
def receipt_lines(prices):
    lines = [f"{name:10} {p:6.2f}" for name, p in prices.items()]
    return lines + [f"{'TOTAL':10} {sum(prices.values()):6.2f}"]

def print_receipt(prices):
    print("\n".join(receipt_lines(prices)))

assert receipt_lines({"tea": 2.5})[-1] == "TOTAL      2.50"
```

`receipt_lines` is pure: the same prices always give the same lines, and it changes nothing. That makes it testable with one comparison, and reusable for writing the receipt to a file, an email or a web page. `print_receipt` is a one-line wrapper with the side effect, too simple to need a test. The general pattern is to build the value, then output it, rather than outputting as you go.

EXERCISE 4.2 A default that remembers

Someone writes `def log(message, when=datetime.now()):` to stamp messages with the time. What's wrong? Fix it.

SOLUTION

The default is evaluated once, when the module is loaded and the `def` runs, so every message logged without an explicit time gets the same timestamp: the moment the program started. It's the mutable-default problem's twin: any default that should be computed fresh each call must be computed inside the function. Use `def log(message, when=None):` and then `when = datetime.now() if when is None else when`. Keeping the parameter has a second benefit: a test can pass a

fixed time and check the output exactly, which is impossible when the function reads the clock itself. Chapter 20 uses the same idea, passing a clock in, to make time-dependent code testable.

EXERCISE 4.3 Functions that make functions

Write `compose(f, g)`, which returns a function that applies `g` and then `f`, and `repeat(f, n)`, which returns a function that applies `f` `n` times. Use them to build a function that adds 10 to a number by adding 1 ten times.

SOLUTION

```
def compose(f, g):
    return lambda x: f(g(x))

def repeat(f, n):
    def apply(x):
        for _ in range(n):
            x = f(x)
        return x
    return apply

add_ten = repeat(lambda x: x + 1, 10)
assert add_ten(5) == 15
assert compose(str, add_ten)(5) == "15"
```

Both return closures: the returned function remembers `f`, `g` and `n` from the call that created it. `repeat` uses a loop rather than composing `f` with itself `n` times, which would also work but build a chain of `n` nested calls. Exercises like this feel abstract at first. They're the core of Composing Programs chapter 1.6, and the habit they build, treating behaviour as something to pass around, pays off whenever code needs to be configurable.

CHECKLIST

- ☐ Can you break a problem into functions that each have one job?
- ☐ Can you tell a pure function from one with side effects, and keep side effects at the edges?
- ☐ Can you explain when default values are evaluated, and avoid mutable defaults?
- ☐ Can you use positional, keyword, keyword-only, `*args` and `**kwargs` parameters?

☐ Can you pass a function as an argument and return one from another function?

GO DEEPER

BOOK Composing Programs, 1.4: Designing Functions John DeNero

BOOK Composing Programs, 1.6: Higher-Order Functions John DeNero

DOCS The Python Tutorial: Defining Functions Python documentation

Recursion and the call stack

3 exercises

A recursive function solves a problem by calling itself on a smaller version of the same problem, until the problem is small enough to answer directly. It's the natural way to handle anything that contains smaller things of the same kind: nested lists, folders inside folders, expressions inside expressions. Recursion seems like magic until you see the mechanism behind it, the call stack, after which it's just function calls. This chapter shows the stack growing and unwinding, counts the cost of solving the same subproblem twice, and shows what happens when the stack runs out.

Base case and recursive case

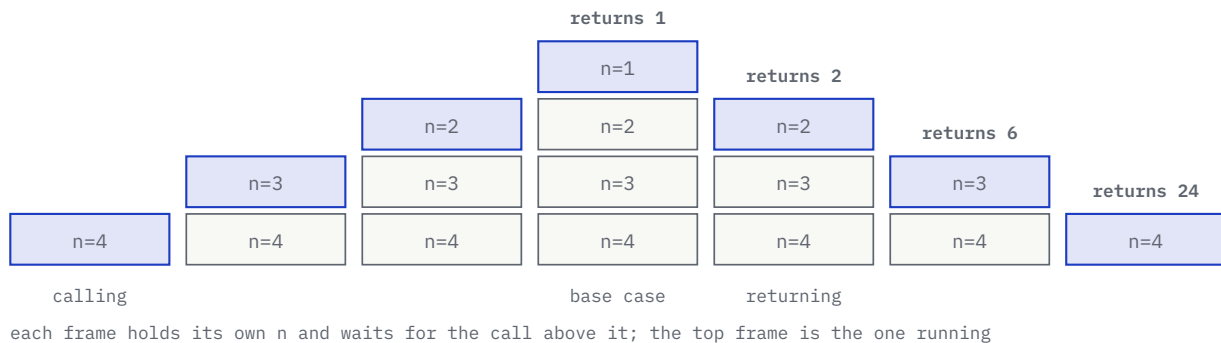
Every recursive function has two parts. The **base case** answers the smallest problems directly, without recursion: the factorial of 1 is 1, the total of a number is the number itself. The **recursive case** breaks a bigger problem into smaller ones of the same kind, solves those by calling the function, and combines the answers: the factorial of n is n times the factorial of $n - 1$. The recursion is correct if three things hold: the base case is right, the recursive case is right *assuming* the smaller calls return correct answers, and every chain of calls eventually reaches a base case.

That second point is the leap of faith that makes recursion manageable. When writing `total(item)` for nested lists, don't trace through every level. Assume `total` already works for each element, and ask only: given the totals of the elements, how do I get the total of the list? Sum them. The function is then correct by the same induction argument a mathematician would use.

The call stack

Each call to a function gets a **frame**: a record holding its parameters and local variables, and the place to return to when it finishes. Frames are kept on the **call stack**. A call pushes a new frame on top; a return pops it and resumes the frame below. Recursion needs nothing more. `factorial(4)` and the `factorial(3)` it calls are separate frames with separate values of `n`, so they don't interfere, and the first can't finish until the one above it returns.

THE CALL STACK DURING FACTORIAL(4)



The stack grows by one frame per call until the base case, then shrinks by one per return. Each frame keeps its own `n` while it waits for the call above it.

The example's first part prints this happening, indented by depth: four calls going down, then four returns coming back up, each multiplying by its own `n`. Python Tutor, a free website, draws the frames for any short program as it runs, and stepping through a recursive function there once is the fastest way to make the mechanism concrete.

EXAMPLE

recursion.py · the call stack growing and unwinding, recursion over nested lists, memoisation, the recursion limit, an explicit stack, and the towers of Hanoi

```
import sys
from functools import cache

# 1. Each call gets its own frame on the call stack. Watch the stack grow, then unwind.
def factorial(n, depth=0):
    pad = " " * depth
    print(f"{pad}factorial({n}) called")
    if n <= 1:
        print(f"{pad}factorial({n}) returns 1")
        return 1
    result = n * factorial(n - 1, depth + 1)
    print(f"{pad}factorial({n}) returns {n} * {result // n} = {result}")
    return result
factorial(4)

# 2. Recursive data invites recursive code: a list can contain lists
def total(item):
    if isinstance(item, list):
        return sum(total(x) for x in item)
    return item
def depth(item):
    if not isinstance(item, list): return 0
    return 1 + max((depth(x) for x in item), default=0)
nested = [1, [2, [3, 4], []], [[5]], 6]
print(f"\nnested = {nested}: total {total(nested)}, nesting depth {depth(nested)}")

# 3. The same subproblem solved again and again, unless you remember the answers
calls = 0
def fib(n):
    global calls; calls += 1
```

```

    return n if n < 2 else fib(n - 1) + fib(n - 2)
@cache
def fib_memo(n):
    global calls; calls += 1
    return n if n < 2 else fib_memo(n - 1) + fib_memo(n - 2)
for f in (fib, fib_memo):
    calls = 0
    print(f"{f.__name__}(25) = {f(25):,} after {calls:}, {calls}")

# 4. Every frame costs stack space, so Python stops runaway recursion
def sum_to(n):
    return 0 if n == 0 else n + sum_to(n - 1)
print(f"\nPython's recursion limit is {sys.getrecursionlimit()} frames")
for n in (500, 5000):
    try: print(f"sum_to({n}) = {sum_to(n):,}")
    except RecursionError: print(f"sum_to({n}): RecursionError, the stack would be {n:}, {frames}
        deep")

# 5. Any recursion can be turned into a loop with a stack you manage yourself
def flatten_recursive(item):
    if not isinstance(item, list): return [item]
    return [x for sub in item for x in flatten_recursive(sub)]
def flatten_iterative(item):
    out, stack = [], [item]
    while stack:
        top = stack.pop()
        if isinstance(top, list): stack.extend(reversed(top)) # reversed keeps left-to-right
            order
        else: out.append(top)
    return out
deep = 0
for _ in range(10_000): deep = [deep] # 10,000 lists, one inside the
    next
print(f"flatten(nested): recursive {flatten_recursive(nested)}, iterative
    {flatten_iterative(nested)}")
try: flatten_recursive(deep)
except RecursionError: print("10,000 levels deep: the recursive version overflows; "
    f"the iterative one returns {flatten_iterative(deep)}")

# 6. Some problems are naturally recursive: the towers of Hanoi
def hanoi(n, source="A", spare="B", target="C"):
    if n == 0: return []
    return hanoi(n - 1, source, target, spare) + [(source, target)] + hanoi(n - 1, spare,
        source, target)
moves = hanoi(3)
print(f"\nHanoi with 3 discs: {len(moves)} moves: {' '.join(a + '>' + b for a, b in moves)}")
print("moves for 1 to 10 discs:", [len(hanoi(n)) for n in range(1, 11)])

```

OUTPUT

```

factorial(4) called
  factorial(3) called
    factorial(2) called
      factorial(1) called
        factorial(1) returns 1
      factorial(2) returns 2 * 1 = 2
    factorial(3) returns 3 * 2 = 6
  factorial(4) returns 4 * 6 = 24

nested = [1, [2, [3, 4], []], [[5]], 6]: total 21, nesting depth 4
fib(25) = 75,025 after 242,785 calls
fib_memo(25) = 75,025 after 26 calls

Python's recursion limit is 1000 frames
sum_to(500) = 125,250
sum_to(5000): RecursionError, the stack would be 5,000 frames deep

```

```
flatten(nested): recursive [1, 2, 3, 4, 5, 6], iterative [1, 2, 3, 4, 5, 6]
10,000 levels deep: the recursive version overflows; the iterative one returns [0]

Hanoi with 3 discs: 7 moves: A>C A>B C>B A>C B>A B>C A>C
moves for 1 to 10 discs: [1, 3, 7, 15, 31, 63, 127, 255, 511, 1023]
```

Recursive data

Recursion fits best when the data is recursive. A list that can contain lists, a folder that can contain folders, an HTML element that contains elements, a JSON value that contains JSON values: for each, the recursive function has one case for the "leaf" kind and one for the "container" kind, and the container case calls the function on each child. The example's `total` and `depth` are five lines each and handle any nesting. Writing them with loops alone is possible but harder, because a loop has to keep track by hand of where it is in the structure, which is exactly what the call stack does for free.

When recursion repeats itself

The obvious recursive definition of Fibonacci numbers, `fib(n) = fib(n - 1) + fib(n - 2)`, is correct and disastrously slow. `fib(25)` made 242,785 calls, because `fib(23)` is computed twice, `fib(22)` three times, and smaller values thousands of times: the number of calls grows exponentially with n . The fix is **memoisation**: remember each answer the first time it's computed, and look it up afterwards. `functools.cache` does this with one line, and the call count fell to 26, one for each value from 0 to 25. Memoisation works for any pure function, which is one more reason to write functions without side effects. It's also the starting point of dynamic programming, which *The Engineer's Workbench* covers in depth.

When the stack runs out

Every frame takes memory, and Python limits the stack depth, to 1,000 frames by default, so that runaway recursion stops with a `RecursionError` instead of crashing the interpreter. `sum_to(500)` worked and `sum_to(5000)` didn't: a recursion as deep as its input is only safe for small inputs. You can raise the limit with `sys.setrecursionlimit`, but that only moves the problem, and too high a limit can crash Python outright when the real stack overflows.

The robust alternative is to manage a stack yourself. `flatten_iterative` keeps a list of items still to process, pops one, and pushes its children if it has any: the same work as the recursive version, with the pending items on a list on the heap instead of frames on the call stack. It flattened ten thousand levels of nesting that made the recursive version overflow. Any recursion can be converted this way, at some cost in clarity, so a good rule is to write recursion where depth is bounded by something small, like the nesting of a document, and loops where it's bounded by the size of the input. Some languages remove the problem for calls in tail position; Python deliberately doesn't, for reasons chapter 14 of *The Engineer's Press* describes.

Some problems are naturally recursive and stay that way. To move a tower of n discs in the towers of Hanoi, move the top $n - 1$ to the spare peg, move the biggest disc, then move the $n - 1$ back on top: three lines that would be very hard to write as a loop. The move counts the example prints, 1, 3, 7, 15 and so on, are $2^n - 1$, which exercise 5.3 proves.

COMMON TRAPS

A recursive case that doesn't shrink the problem, or a missing base case: infinite recursion, then `RecursionError`.

Recurring once per element of a long list, which hits the depth limit at about a thousand elements.

Solving the same subproblem repeatedly. Memoise pure recursive functions with `functools.cache`.

Raising the recursion limit instead of converting deep recursion to a loop.

Tracing every level by hand instead of trusting the recursive call and checking the combine step.

EXERCISES

EXERCISE 5.1 Sizing a folder tree

A folder is represented as a dict whose values are either file sizes, integers, or nested dicts, which are subfolders. Write a recursive `size(folder)` that returns the total size, and `largest_file(folder)` that returns the size of the biggest file anywhere inside.

SOLUTION

```
def size(folder):
    return sum(size(v) if isinstance(v, dict) else v for v in folder.values())

def largest_file(folder):
    return max((largest_file(v) if isinstance(v, dict) else v for v in
                folder.values()),
               default=0)

home = {"notes.txt": 120, "photos": {"a.jpg": 3000, "b.jpg": 4500, "old": {}},
        "cv.pdf": 800}
assert size(home) == 8420 and largest_file(home) == 4500
```

Each function has a leaf case, a file, whose answer is its own size, and a container case, a folder, which combines its children's answers: by adding for `size`, by taking the maximum for `largest_file`. The empty folder `old` is a base case too: `sum` of nothing is 0, and `max` needs `default=0` or it would raise an error on an empty sequence. Real folder trees are walked exactly like this; `os.walk` in the standard library does the iteration for you on real directories.

EXERCISE 5.2 How many calls

Show that the naive `fib(n)` makes $2 \times \text{fib}(n + 1) - 1$ calls, which checks against the example's 242,785 for $n = 25$. Why does the memoised version make exactly 26?

SOLUTION

Let $C(n)$ be the number of calls `fib(n)` makes, itself included. $C(0) = C(1) = 1$, and for larger n , $C(n) = 1 + C(n - 1) + C(n - 2)$. Adding 1 to both sides gives $C(n) + 1 = (C(n - 1) + 1) + (C(n - 2) + 1)$, which is the Fibonacci recurrence, starting from 2 and 2. So $C(n) + 1 = 2 \times \text{fib}(n + 1)$, and for $n = 25$ that's $2 \times 121,393 - 1 = 242,785$. The memoised version computes each value from 0 to 25 once. Every later request is answered from the cache without running the function's body, and the example's counter is inside the body, so it counts only those 26. 26 values, 26 calls: the exponential tree has collapsed into a line.

EXERCISE 5.3 Why Hanoi takes $2^n - 1$ moves

Prove that `hanoi(n)` makes exactly $2^n - 1$ moves. How long would 64 discs take at one move per second?

SOLUTION

Let $M(n)$ be the number of moves. $M(0) = 0$. For $n \geq 1$, the function moves $n - 1$ discs, then one disc, then $n - 1$ discs, so $M(n) = 2M(n - 1) + 1$. Suppose $M(n - 1) = 2^{n-1} - 1$. Then $M(n) = 2(2^{n-1} - 1) + 1 = 2^n - 1$, and since it holds for 0, by induction it holds for all n . The proof has exactly the shape of the function: a base case and a step that assumes the smaller case. With 64 discs, $2^{64} - 1$ is about 1.8×10^{19} moves, which at one per second is about 585 billion years. No better algorithm exists, either: the largest disc must move at least once, and before it can, all the others must be stacked on the spare peg, which gives the same recurrence as a lower bound.

CHECKLIST

- ☐ Can you write a recursive function with a correct base case and a shrinking recursive case?
- ☐ Can you draw the call stack for a recursive call and explain what each frame holds?
- ☐ Can you write recursion over nested data such as lists of lists or folder trees?
- ☐ Can you spot repeated subproblems and fix them with memoisation?
- ☐ Can you convert a deep recursion to a loop with an explicit stack?

GO DEEPER

BOOK [Composing Programs, 1.7: Recursive Functions](#) John DeNero

BOOK [SICP, 1.2: Procedures and the Processes They Generate](#) Abelson & Sussman

TOOL [Python Tutor: visualise code as it runs](#) Philip Guo

PART II

Data

Four chapters on the shapes data takes inside a program and on its way in and out. Lists and tuples keep things in order; dictionaries and sets find things by key. Each makes some operations nearly free and others expensive, and knowing which is the difference between a program that handles a million records and one that stalls at ten thousand. Text needs a chapter of its own, because what you see on screen and what's stored in a file are different things. And files bring all of it together: reading data that's messier than you'd like, and writing results other programs can read.

Lists and tuples

3 exercises

A list is an ordered collection you can change; a tuple is an ordered collection you can't. Both are sequences, so they share indexing, slicing and unpacking, and both are built on an array, a single block of memory with items side by side. That layout makes some operations instant and others slow in proportion to the length. This chapter covers the operations you'll use every day, then counts the work behind `append`, `insert` and `in`, so you can predict which code will still be fast with a million items.

Sequences

A **list** is written in square brackets, `["Mercury", "Venus", "Earth"]`, and holds any values, of any types, in order. Items are numbered from 0, and negative indexes count from the end, so `planets[-1]` is the last item. A **slice**, `planets[1:3]`, is a new list of the items from index 1 up to, but not including, index 3: the same half-open convention as `range` and chapter 3's binary search, which makes `xs[:k]` and `xs[k:]` split a list cleanly with nothing lost or repeated. A third number in a slice is a step, so `xs[::2]` takes every other item and `xs[::-1]` is the list reversed.

Lists change in place. `append` adds to the end, `extend` adds several, `insert(i, x)` puts an item at position `i`, `pop()` removes and returns the last item, `remove(x)` removes the first item equal to `x`, and `sort()` sorts. A **list comprehension** builds a new list from an old one in one expression: `[x * x for x in nums if x % 2 == 0]` is the squares of the even numbers. Comprehensions are usually clearer than the equivalent loop with `append`, as long as they fit on a line or two.

A **tuple**, written with parentheses, `(3, 4)`, is a sequence that can't be changed after it's made. Use one for a fixed group of values that belong together, like a point or a date, and a list for a collection of similar things whose length varies. Because tuples can't change, they can be dictionary keys and set members, which lists can't, as the example shows. Tuples also make **unpacking** natural: `x, y = point` splits a pair into two names, `first, *middle, last = planets` gathers the rest into a list, and a function can return several values as a tuple, `return lowest, highest`, that the caller unpacks.

EXAMPLE

lists.py · indexing, slicing, unpacking and tuples, then the cost of growing a list and searching it, stable sorting, and the bug of changing a list while looping over it

```
# 1. Lists are ordered and changeable; tuples are ordered and fixed
planets = ["Mercury", "Venus", "Earth", "Mars", "Jupiter"]
print(f"planets[0] = {planets[0]!r}, planets[-1] = {planets[-1]!r}, planets[1:3] = {planets[1:3]!r}")
print(f"planets[::2] = {planets[::2]!r}, planets[::-1][2] = {planets[::-1][2]!r}")
```

```

first, *middle, last = planets                                # unpacking, with the rest gathered up
print(f"first = {first!r}, middle = {middle}, last = {last!r}")
point = (3, 4)
try: point[0] = 5
except TypeError as e: print(f"point[0] = 5 -> TypeError: {e}")
distance = {(0, 0): "origin", point: "five away"} # tuples can be dict keys; lists can't
try: distance[[1, 2]] = "nope"
except TypeError as e: print(f"a list as a key -> TypeError: {e}")

# 2. Why append is cheap and insert(0) is not: a list is an array that grows by reallocating
print()
class DynamicArray:
    """Counts the element copies a list does as it grows, for a given growth policy."""
    def __init__(self, grow):
        self.grow, self.capacity, self.size, self.copies, self.shifts = grow, 1, 0, 0, 0
    def append(self):
        if self.size == self.capacity:                                # full: allocate a bigger block, copy
            all
            self.copies += self.size
            self.capacity = self.grow(self.capacity)
            self.size += 1
    def insert_front(self):
        self.append()
        self.shifts += self.size - 1                                # every existing item moves up one place
for n in (1_000, 100_000):
    by_one, doubling = DynamicArray(lambda c: c + 1), DynamicArray(lambda c: 2 * c)
    for _ in range(n): by_one.append(); doubling.append()
    print(f"{n}>7,} appends: growing by 1 copies {by_one.copies:>13,} items; "
          f"doubling copies {doubling.copies:>7,}, {doubling.copies / n:.2f} per append")
front = DynamicArray(lambda c: 2 * c)
for _ in range(10_000): front.insert_front()
print(f"10,000 inserts at the front shift {front.shifts:,} items, {front.shifts // 10_000:,}
      per insert")

# 3. 'in' on a list checks items one by one
comparisons = 0
class Counted:
    def __init__(self, v): self.v = v
    def __eq__(self, other):
        global comparisons; comparisons += 1
        return self.v == other.v
items = [Counted(i) for i in range(10_000)]
Counted(9_999) in items
print(f"\nfinding the last of 10,000 items with 'in': {comparisons:,} comparisons")

# 4. Sorting is stable: items that compare equal keep their original order
people = [("Ana", 31), ("Ben", 25), ("Cy", 31), ("Di", 25)]
by_age = sorted(people, key=lambda p: p[1])
print(f"\nsorted by age: {by_age} (Ben before Di, Ana before Cy, as in the input)")
print(f"oldest first, then by name: {sorted(people, key=lambda p: (-p[1], p[0]))}")

# 5. A classic bug: changing a list while looping over it
nums = [2, 4, 5, 6, 8]
for x in nums:
    if x % 2 == 0: nums.remove(x)                                # the loop's position skips the next item
print(f"\nremoving evens while iterating leaves {nums}")
nums = [2, 4, 5, 6, 8]
nums = [x for x in nums if x % 2 != 0]                            # build a new list instead
print(f"building a new list instead leaves {nums}")

```

OUTPUT

```

planets[0] = 'Mercury', planets[-1] = 'Jupiter', planets[1:3] = ['Venus', 'Earth']
planets[:2] = ['Mercury', 'Earth', 'Jupiter'], planets[:-1][2] = ['Jupiter', 'Mars']
first = 'Mercury', middle = ['Venus', 'Earth', 'Mars'], last = 'Jupiter'
point[0] = 5 -> TypeError: 'tuple' object does not support item assignment

```

```

a list as a key -> TypeError: cannot use 'list' as a dict key (unhashable type: 'list')

1,000 appends: growing by 1 copies      499,500 items; doubling copies  1,023, 1.02 per
append
100,000 appends: growing by 1 copies 4,999,950,000 items; doubling copies 131,071, 1.31 per
append
10,000 inserts at the front shift 49,995,000 items, 4,999 per insert

finding the last of 10,000 items with 'in': 10,000 comparisons

sorted by age: [('Ben', 25), ('Di', 25), ('Ana', 31), ('Cy', 31)] (Ben before Di, Ana before
Cy, as in the input)
oldest first, then by name: [('Ana', 31), ('Cy', 31), ('Ben', 25), ('Di', 25)]

removing evens while iterating leaves [4, 5, 8]
building a new list instead leaves [5]

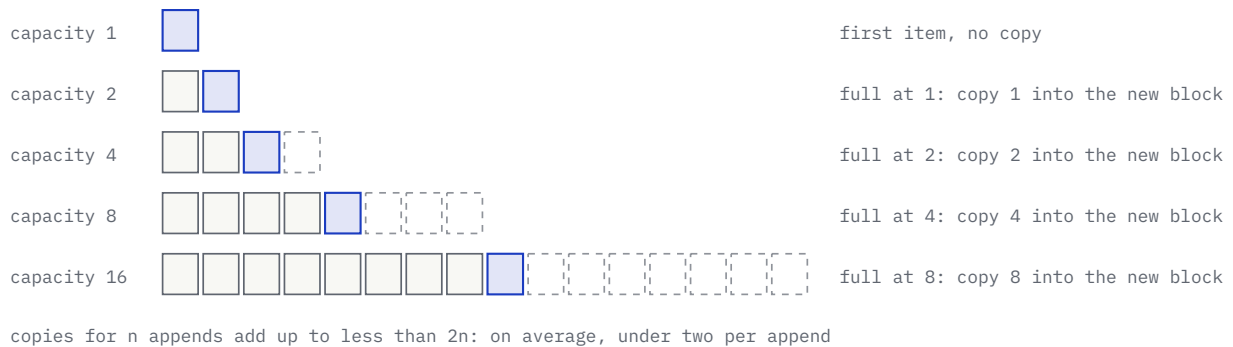
```

What operations cost

A list stores references to its items in one contiguous array, so finding item `i` is a single calculation, the start of the array plus `i` slots, however long the list is. That's why `xs[i]` and `xs[i] = v` take the same time for a list of ten items or ten million. Programmers write this as $O(1)$, constant time. Operations whose work grows in proportion to the length n are $O(n)$, linear time, and ones that do a linear amount of work for each of n items are $O(n^2)$, which is fine for a hundred items and hopeless for a million.

Appending is the interesting case. The array has a fixed **capacity**; when it's full, the list allocates a bigger block and copies everything across. If it grew by one slot at a time, every append would copy the whole list, and the example's 100,000 appends would copy nearly five billion items. Instead it grows by a constant *factor*. With doubling, a copy of k items happens only after k cheap appends, so the copies add up to less than $2n$ in total: the example measured 1.31 copies per append. On average each append is $O(1)$, which is called **amortised** constant time. CPython grows lists by about an eighth each time rather than doubling, which wastes less memory, and any constant growth factor gives the same $O(1)$ average.

APPENDING, WITH THE CAPACITY DOUBLED WHENEVER IT'S FULL



INSERTING AT THE FRONT INSTEAD



Why `append` is cheap and `insert(0, x)` isn't. Doubling makes copying rare, so its cost averages out; inserting at the front moves every item, every time.

Inserting or removing at the front is the opposite: every other item must move one place, so `insert(0, x)` and `pop(0)` are $O(n)$, and 10,000 front inserts shifted almost fifty million items. If you need to add and remove at both ends, as a queue does, use `collections.deque`, which does both in $O(1)$. And `x in xs` must compare `x` with each item until it finds a match, so it's $O(n)$: finding the last of 10,000 items took 10,000 comparisons. Doing that inside a loop over another list of 10,000 is 100 million comparisons, which is where programs that work in testing stall in production. Chapter 7's sets do the same test in $O(1)$.

Sorting

`sorted(xs)` returns a new sorted list and `xs.sort()` sorts in place; both take `key=`, a function that computes what to compare, and `reverse=True`. Python's sort, Timsort, takes $O(n \log n)$ time, and it's **stable**: items whose keys are equal stay in their original order. Stability is what makes multi-level sorting simple. To sort people by age and then by name within an age, use a tuple key, `key=lambda p: (p[1], p[0])`, or sort by name first and then by age, and the second sort keeps the names in order. Negating a number in the key, as the example does, reverses that part of the order only.

Don't change a list while looping over it

The example's last part removes even numbers from `[2, 4, 5, 6, 8]` inside a `for` loop over the same list and is left with `[4, 5, 8]`. The loop keeps an index into the list. Removing an item shifts everything after it one place left, so the item that slides into the current position is skipped. Exercise 6.2 traces it. The reliable fix is to build a new list with a comprehension, or loop over a copy, `for x in xs[:]`. The same rule applies to dicts and sets, where Python detects the change and raises an error instead of silently skipping.

COMMON TRAPS

Using a list as a queue with `pop(0)` or `insert(0, x)`. Use `collections.deque`.

Testing membership in a big list inside a loop, turning an $O(n)$ program into $O(n^2)$. Use a set.

Removing items from a list while iterating over it. Build a new list instead.

Writing `xs = xs.sort()` and losing the list to `None`.

Using a list where a tuple belongs, for a fixed record, and then being unable to use it as a dict key.

EXERCISES

EXERCISE 6.1 A queue

A print queue receives jobs at the back and serves them from the front. Implement it with a list and with `collections.deque`. For 100,000 jobs, roughly how many item moves does each version make?

SOLUTION

```
from collections import deque

jobs = deque()
for n in range(100_000): jobs.append(f"job {n}")      # O(1) at the back
while jobs: jobs.popleft()                          # O(1) at the front
```

With a list, `append` is cheap, but each `pop(0)` shifts every remaining job one place. Serving 100,000 jobs moves $99,999 + 99,998 + \dots + 1$ items, about five billion, which takes seconds instead of milliseconds. A `deque`, a double-ended queue, is built from linked blocks of items so that both ends can grow and shrink without moving anything else; its `append`, `appendleft`, `pop` and `popleft` are all $O(1)$, and it makes no item moves at all here. The price is that indexing into the middle of a deque is $O(n)$, so it's for queues and stacks, not for random access.

EXERCISE 6.2 The skipped items

Trace the loop that removes evens from `[2, 4, 5, 6, 8]` while iterating, writing the list and the loop's position after each step, and explain why the result is `[4, 5, 8]`.

SOLUTION

The `for` loop keeps a hidden position, starting at 0. Position 0 holds 2, which is even, so it's removed, and the list becomes `[4, 5, 6, 8]`. The position moves to 1, which now holds 5: the 4 slid into position 0 and is never looked at. 5 is odd and stays. Position 2 holds 6, which is removed, giving `[4, 5, 8]`. Position 3 is past the end of a three-item list, so the loop stops, and 8 was skipped the same way 4 was. Every removal skips the item that follows. That's why the result depends on where the evens are: `[1, 2, 3, 4]` would happen to come out right, which is the worst kind of bug, one that passes some tests. A comprehension builds the new list without touching the one being read.

EXERCISE 6.3 Returning two values

Write `span(nums)`, which returns the smallest and largest numbers in one pass, and call it with unpacking. Why is a tuple the right return type here, rather than a list or two separate functions?

SOLUTION

```
def span(nums):
    lo = hi = nums[0]
    for x in nums[1:]:
        if x < lo: lo = x
        elif x > hi: hi = x
    return lo, hi

low, high = span([4, -2, 9, 0])
assert (low, high) == (-2, 9)
```

The result is a fixed pair of two different things, a minimum and a maximum, whose positions have meanings, which is exactly what a tuple is for; a list would suggest a collection of similar items that might grow. One pass over the data does the work of two, which matters when the data is large or can only be read once, as with a file or a network stream. Unpacking at the call site names both parts immediately, and if the number of names doesn't match the tuple's length, Python raises an error instead of silently misassigning. For results with more than three parts, a named tuple or a dataclass, chapter 13, makes the fields self-describing.

CHECKLIST

- ☐ Can you index, slice and unpack sequences, including negative indexes and steps?
- ☐ Can you choose between a list and a tuple, and explain why only tuples can be dict keys?
- ☐ Can you explain why `append` is $O(1)$ on average while `insert(0, x)` is $O(n)$?
- ☐ Can you sort by several keys using a key function and stability?
- ☐ Can you spot and fix code that changes a list while iterating over it?

GO DEEPER

DOCS [The Python Tutorial: Data Structures](#) Python documentation

WIKI [TimeComplexity: the cost of operations on Python's built-in types](#) Python wiki

DOCS [Sorting Techniques](#) Andrew Dalke & Raymond Hettinger

Dictionaries and sets

3 exercises

A dictionary finds a value by its key, and a set answers whether it contains something, and both do it in about the same time whether they hold ten items or ten million. That one property is behind a large share of all fast programs. The mechanism is a hash table: turn the key into a number, use the number to pick a slot, and look only there. This chapter shows dicts and sets at work, builds a small hash table to show where the speed comes from, and explains the rule that makes it work: equal keys must have equal hashes.

Dictionaries

A **dict** maps **keys** to **values**: `{"apples": 12, "pears": 0}`. `stock["apples"]` looks a value up, `stock["figs"] = 7` adds or replaces one, and `del stock["pears"]` removes one. Looking up a missing key raises `KeyError`, which is what you want when the key should be there. When a missing key is normal, `stock.get("kiwis", 0)` returns a default instead. `for key in d` loops over the keys, and `d.items()` gives key-value pairs. Since Python 3.7, dicts keep their keys in the order they were first inserted, which makes output predictable and lets `dict.fromkeys(items)` remove duplicates while keeping the first occurrence of each.

Two variations from the `collections` module save a great deal of code. A `Counter` counts things: `Counter(words).most_common(3)` is a word-frequency table in one line. A `defaultdict(list)` creates an empty list the first time a key is used, so grouping items is `groups[key].append(item)` with no check for whether the key exists yet. Both are ordinary dicts underneath.

Sets

A **set** is a collection of unique values with no order: `{1, 2, 3}`, or `set(items)` to build one from a list. Its main job is fast membership testing, `x in s`, and it supports the operations of mathematics: `a | b` is the union, `a & b` the intersection, `a - b` the difference. Comparing sets ignores order and duplicates, so `set("listen") == set("silent")` is true. Use a set whenever the question is "have I seen this?" or "is this allowed?", and especially to replace `x in some_list` inside a loop.

EXAMPLE

`dicts.py` · dicts, Counter, defaultdict and sets; a small hash table with a weak hash and a good one; and why keys need equal hashes when they're equal

```
from collections import Counter, defaultdict
```

```

# 1. A dict maps keys to values; a set holds unique keys. Both remember insertion order
(dict).
stock = {"apples": 12, "pears": 0}
stock["figs"] = 7
print(f"stock = {stock}; stock.get('kiwis', 0) = {stock.get('kiwis', 0)}")
try: stock["kiwis"]
except KeyError as e: print(f"stock['kiwis'] -> KeyError: {e}")
in_stock = {name for name, n in stock.items() if n > 0}
print(f"in stock: {sorted(in_stock)}; 'pears' in stock: {'pears' in stock} (the key exists,
      with 0)")

words = "the cat and the hat and the bat".split()
print(f"\nCounter: {Counter(words).most_common(2)}")
by_letter = defaultdict(list)
for w in words: by_letter[w[0]].append(w)
print(f"grouped by first letter: {dict(by_letter)}")
print(f"unique, in first-seen order: {list(dict.fromkeys(words))}")
a, b = set("listen"), set("silent")
print(f"set('listen') == set('silent'): {a == b}; "
      f"{1, 2, 3} & {2, 3, 4} = {(1, 2, 3) & (2, 3, 4)}, | = {(1, 2, 3) | (2, 3, 4)}, "
      f"- = {(1, 2, 3) - (2, 3, 4)}")

# 2. How a dict finds a key without searching: hash it to a slot, then probe a few slots
def weak_hash(s):
    fixed
    h = 0
    for ch in s: h = (h * 31 + ord(ch)) % 2**32
    return h
def fnv_hash(s):
    # FNV-1a: mixes every character into every bit
    h = 2166136261
    for ch in s: h = ((h ^ ord(ch)) * 16777619) % 2**32
    return h
class HashTable:
    def __init__(self, hash_fn, size=8):
        self.hash, self.slots, self.count, self.probes = hash_fn, [None] * size, 0, 0
    def _find(self, key):
        i = self.hash(key) % len(self.slots)
        while True:
            self.probes += 1
            if self.slots[i] is None or self.slots[i][0] == key: return i
            i = (i + 1) % len(self.slots) # occupied by another key: try the next slot
    def put(self, key, value):
        if (self.count + 1) * 3 > len(self.slots) * 2: # keep it under two-thirds full
            old = [s for s in self.slots if s]
            self.slots, self.count = [None] * (len(self.slots) * 4), 0
            for k, v in old: self.put(k, v)
        i = self._find(key)
        if self.slots[i] is None: self.count += 1
        self.slots[i] = (key, value)
    def get(self, key):
        slot = self.slots[self._find(key)]
        return slot[1] if slot else None
names = [f"user{i}" for i in range(10_000)]
print()
for hash_fn in (weak_hash, fnv_hash):
    table = HashTable(hash_fn)
    for i, name in enumerate(names): table.put(name, i)
    table.probes = 0
    assert all(table.get(name) == i for i, name in enumerate(names))
    print(f"10,000 keys in {len(table.slots):,} slots with {hash_fn.__name__}: "
          f"{table.probes / len(names):.2f} probes per lookup")
print(f"the same lookups in a list would average {sum(range(1, 10_001)) / 10_000:.1f}
      comparisons")

# 3. Keys must be hashable, and equal keys must hash equally

```

```

class Point:
    def __init__(self, x, y): self.x, self.y = x, y
    def __eq__(self, other): return (self.x, self.y) == (other.x, other.y)
try: {Point(1, 2): "a"}
except TypeError as e: print(f"\nPoint with __eq__ but no __hash__ -> TypeError: {e}")
class BadPoint(Point):
    def __hash__(self): return id(self)      # breaks the rule: equal points, different hashes
places = {BadPoint(1, 2): "home"}
print(f"BadPoint(1, 2) in places: {BadPoint(1, 2) in places} (it's equal, but hashed elsewhere)")
class GoodPoint(Point):
    def __hash__(self): return hash((self.x, self.y))
places = {GoodPoint(1, 2): "home"}
print(f"GoodPoint(1, 2) in places: {GoodPoint(1, 2) in places}")

```

OUTPUT

```

stock = {'apples': 12, 'pears': 0, 'figs': 7}; stock.get('kiwis', 0) = 0
stock['kiwis'] -> KeyError: 'kiwis'
in stock: ['apples', 'figs']; 'pears' in stock: True (the key exists, with 0)

Counter: [('the', 3), ('and', 2)]
grouped by first letter: {'t': ['the', 'the', 'the'], 'c': ['cat'], 'a': ['and', 'and'], 'h':
    ['hat'], 'b': ['bat']}
unique, in first-seen order: ['the', 'cat', 'and', 'hat', 'bat']
set('listen') == set('silent'): True; {1, 2, 3} & {2, 3, 4} = {2, 3}, | = {1, 2, 3, 4}, - = {1}

10,000 keys in 32,768 slots with weak_hash: 8.12 probes per lookup
10,000 keys in 32,768 slots with fnv_hash: 1.18 probes per lookup
the same lookups in a list would average 5,000.5 comparisons

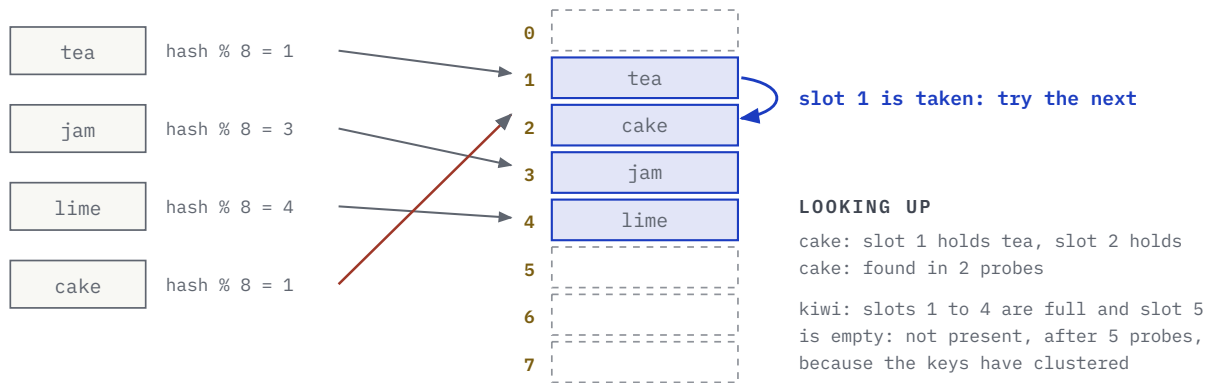
Point with __eq__ but no __hash__ -> TypeError: cannot use 'Point' as a dict key (unhashable
    type: 'Point')
BadPoint(1, 2) in places: False (it's equal, but hashed elsewhere)
GoodPoint(1, 2) in places: True

```

How a hash table finds things

A dict is an array of **slots**. To store a key, Python computes the key's **hash**, an integer derived from its contents, and uses it to choose a slot, for instance the hash modulo the number of slots. To find the key later, it computes the same hash and goes to the same slot. No searching is needed, whatever the dict's size. When two keys land on the same slot, a **collision**, the second goes to another slot found by a fixed rule; the example's table uses the simplest rule, **linear probing**, which tries the next slot along. A lookup then checks a few slots, comparing keys, until it finds the key or an empty slot, which means the key isn't there.

KEYS, HASHED TO SLOTS (FNV-1A, 8 SLOTS)



A collision handled by linear probing, with real FNV-1a hashes. Lookups stay short as long as keys are spread out and the table has room; clusters of neighbouring keys make them longer.

Two things keep lookups short. The table must have room: the example's table grows when it's two-thirds full, so most slots near any key are empty and a probe sequence ends quickly. And the hash must spread keys evenly. The example compares two hash functions on 10,000 keys named `user0` to `user9999`. The weak one, which multiplies by 31 and adds each character, gives similar keys similar hashes, so they land in neighbouring slots and form long clusters: 8.12 probes per lookup. FNV-1a mixes every character into every bit of the result, so similar keys land far apart: 1.18 probes. Either way, compare that with the 5,000 comparisons an average lookup in a list would need.

Python's real dict is more refined. It probes in a pseudo-random order rather than linearly, which avoids clusters even with imperfect hashes, and keeps a compact array of entries in insertion order alongside the slots, which is how dicts remember their order. Python also deliberately **randomises** the hashes of strings with a secret chosen when each program starts, so that an attacker can't craft thousands of keys that all collide and make a web server's dict lookups slow. That's why `hash("tea")` changes between runs, and why the example uses its own fixed hash functions to print the same numbers every time.

The rule for keys

A hash table only works if equal keys have equal hashes, since the hash decides where to look. Python enforces part of this. Built-in immutable types, numbers, strings and tuples of them, are hashable and follow the rule. Mutable built-ins such as lists and dicts aren't hashable at all, because a key whose contents changed after it was stored would have a different hash and be lost in the wrong slot. And a class that defines `__eq__` to compare contents loses its default `__hash__`, so that it can't silently break the rule, which is the `TypeError` the example shows.

If you define `__hash__` yourself, it must agree with `__eq__`. The example's `BadPoint` hashes by identity while comparing by contents, so two equal points usually hash to different slots, and `BadPoint(1, 2) in places` is false even though an equal point is right

there. The correct version hashes the same tuple that `__eq__` compares. In practice you rarely write either method by hand: a frozen dataclass, chapter 13, generates a matching pair.

COMMON TRAPS

Searching a list in a loop when a set or dict lookup would be $O(1)$.

Using `d[key]` where the key may be missing, or `d.get` where it must be present and a missing key is a bug.

Defining `__eq__` without a matching `__hash__`, or hashing on fields that can change.

Relying on `hash()` of a string being the same between runs. It's randomised on purpose.

Expecting a set to keep order. Use `dict.fromkeys` to remove duplicates in order.

EXERCISES

EXERCISE 7.1 Finding duplicates fast

Write a function that returns the email addresses appearing more than once in a list, ignoring case. Make it $O(n)$, and explain why the obvious version with `emails.count(e)` isn't.

SOLUTION

```
from collections import Counter

def duplicates(emails):
    counts = Counter(e.lower() for e in emails)
    return sorted(e for e, n in counts.items() if n > 1)

emails = ["a@x.com", "B@x.com", "A@X.com", "c@x.com", "b@x.com"]
assert duplicates(emails) == ["a@x.com", "b@x.com"]
```

`Counter` makes one pass, and each increment is a dict operation, $O(1)$ on average, so the whole function is $O(n)$ plus the sorting of the small result. The version `[e for e in emails if emails.count(e) > 1]` calls `count`, which scans the whole list, once for every email: n scans of n items, $O(n^2)$. For 100,000 addresses that's ten billion comparisons against 100,000 dict updates. Lower-casing before counting matters too:

email addresses are compared case-insensitively in practice, and normalising the key is the standard way to make a dict treat "A@X.com" and "a@x.com" as one.

EXERCISE 7.2 A key that changes

Suppose Python allowed lists as dict keys. You store `d[k] = "value"` with `k = [1, 2]`, then do `k.append(3)`. Explain what would go wrong with `d[k]` and with `d[[1, 2]]`.

SOLUTION

The entry was stored in the slot chosen by the hash of `[1, 2]`. After the append, `d[k]` computes the hash of `[1, 2, 3]`, which almost certainly picks a different slot, so the lookup finds nothing there and reports a missing key, although the entry is in the dict, holding that very list. `d[[1, 2]]` computes the original hash and goes to the right slot, but the key stored there is now `[1, 2, 3]`, which isn't equal to `[1, 2]`, so that lookup fails too. The entry is unreachable by any key. That's why Python refuses mutable built-ins as keys, and why your own classes should only be hashable if the fields their hash uses can't change: freeze them, or hash on an identifier that never changes.

EXERCISE 7.3 How full is too full

Modify the example's table so it never grows, give it 16,384 slots, and fill it to 50%, 75%, 90% and 98% with the `user` keys. Measure the average probes for keys that are present and for keys that aren't. What does that suggest about why dicts keep spare capacity?

SOLUTION

With FNV-1a and linear probing, lookups of present keys averaged 1.46, 2.49, 5.13 and 47.97 probes at the four levels, and lookups of missing keys 2.62, 8.54, 49.5 and over 3,000. A missing key must probe until it finds an empty slot, and as the table fills, the runs of occupied slots join into long clusters, so the cost explodes near full. The measurements match the textbook formulas for linear probing, about $\frac{1}{2}(1 + 1/(1 - a))$ for a hit and $\frac{1}{2}(1 + 1/(1 - a)^2)$ for a miss, where a is the fraction of slots in use: 2.5 and 8.5 at 75%, 5.5 and 50.5 at 90%. Keeping the table at most two-thirds full, as the example and CPython do, trades memory, at least a third of the slots always empty, for lookups that stay at one or two probes whatever happens.

CHECKLIST

- ☐ Can you use dicts, `get`, `Counter` and `defaultdict` for counting and grouping?
- ☐ Can you use sets for membership and for union, intersection and difference?
- ☐ Can you explain how a hash table finds a key without searching, and what a collision is?
- ☐ Can you explain why a good hash function and spare capacity keep lookups fast?
- ☐ Can you state the rule linking `__eq__` and `__hash__`, and why lists can't be keys?

GO DEEPER

DOCS collections: `Counter`, `defaultdict`, `deque` and friends [Python documentation](#)

TALK Modern Python Dictionaries: A confluence of a dozen great ideas [Raymond Hettinger, PyCon 2017](#)

DOCS The data model: `object.__hash__` [Python documentation](#)

Strings, Unicode and UTF-8

3 exercises

*Inside a Python program, text is a sequence of characters. In a file or on a network, it's a sequence of bytes, and an **encoding** is the rule that converts one to the other. Most text bugs, the `Ã©` where an `é` should be, the crash on a customer's name, the emoji that counts as three characters, come from confusing the two or from guessing the encoding. This chapter separates characters from bytes, encodes UTF-8 by hand to show why it won, and covers the cases where what you see on screen isn't what `len` counts.*

Code points

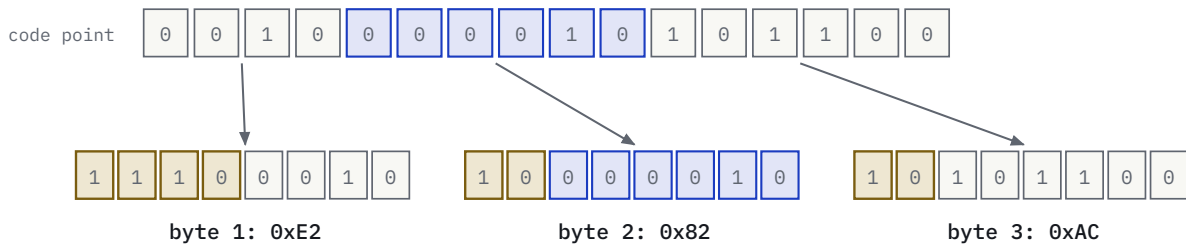
Unicode gives every character in every writing system a number, its **code point**, written `U+` followed by hexadecimal: `A` is `U+0041`, `é` `U+00E9`, `€` `U+20AC`, `😄` `U+1F600`. There are nearly 160,000 assigned characters, out of a space of 1,114,112 possible code points. A Python `str` is a sequence of code points. `ord("€")` gives the number, `chr(8364)` gives the character back, and the `unicodedata` module knows each character's official name. Code points are abstract numbers. Nothing about them says how many bytes a character takes in a file; that's the encoding's job.

Encodings, and why UTF-8 won

For decades, each region used its own encoding. **ASCII** covered English in 7 bits, 128 characters. Latin-1 and Windows-1252 used the eighth bit for Western European accents; other code pages did the same for Greek, Cyrillic or Hebrew, using the same byte values for different letters. A file was only readable if you knew which one it used, and the file didn't say. Unicode replaced the character sets, and of its encodings, **UTF-8** replaced the rest. It was designed by Ken Thompson and Rob Pike in 1992, and today it's used by about 98% of web pages.

UTF-8 uses one to four bytes per code point. Code points up to `U+007F`, which are exactly ASCII, take one byte with the same value, so every ASCII file is already valid UTF-8, which made adopting it painless. Larger code points are split into groups of bits spread across two, three or four bytes, each marked by its leading bits: the first byte of a multi-byte character says how many bytes follow, and every continuation byte starts with `10`. That design means a program can find character boundaries anywhere in a stream, and no character's bytes can appear inside another's.

'€' IS CODE POINT U+20AC: 16 BITS, WHICH NEED THE 3-BYTE FORM



gold: header bits. 1110 means 'first of three bytes'; 10 means 'a continuation byte'

1 byte	U+0000 to U+007F	0xxxxxxx	ASCII, unchanged
2 bytes	U+0080 to U+07FF	110xxxxx 10xxxxxx	most accented Latin, Greek, Cyrillic
3 bytes	U+0800 to U+FFFF	1110xxxx 10xxxxxx 10xxxxxx	most other scripts, €
4 bytes	U+10000 up	11110xxx 10xxxxxx (x3)	emoji, rarer scripts

How UTF-8 stores €. The code point's bits are split into groups and placed after header bits that say how long the character is, so any byte tells you whether it starts a character or continues one.

The example's `utf8` function implements those rules with shifts and masks, and its output matches Python's own `encode("utf-8")` for all four sizes. It's worth reading slowly once: `cp >> 6` drops the low six bits, `& 0x3F` keeps only the low six bits, and `0x80 |` adds the continuation header. After that, you can read the bytes of any UTF-8 text by eye.

EXAMPLE

strings.py · code points, a hand-written UTF-8 encoder checked against Python's, mojibake and decoding errors, normalisation, and characters made of several code points

```
import unicodedata

# 1. A str is a sequence of code points: numbers that Unicode assigns to characters
for ch in ("A", "é", "€", "🍷"):
    print(f"{ch!r}: code point U+{ord(ch):04X} ({ord(ch):,}), name {unicodedata.name(ch)}")

# 2. On disk and on the wire, text is bytes. UTF-8 uses 1 to 4 bytes per code point.
def utf8(cp):
    """Encode one code point by hand, the way UTF-8 is defined."""
    if cp < 0x80: return bytes([cp]) # 0xxxxxxx
    if cp < 0x800: return bytes([0xC0 | cp >> 6, 0x80 | cp & 0x3F]) # 110xxxxx 10xxxxxx
    if cp < 0x10000:
        return bytes([0xE0 | cp >> 12, 0x80 | cp >> 6 & 0x3F, 0x80 | cp & 0x3F])
    return bytes([0xF0 | cp >> 18, 0x80 | cp >> 12 & 0x3F, 0x80 | cp >> 6 & 0x3F, 0x80 | cp & 0x3F])

print()
for ch in ("A", "é", "€", "🍷"):
    mine, python = utf8(ord(ch)), ch.encode("utf-8")
    bits = " ".join(f"{b:08b}" for b in mine)
    unit = "bytes" if len(mine) > 1 else "byte"
    print(f"{ch!r}: {len(mine)} {unit} {bits:<36} same as Python: {mine == python}")

text = "naïve café, 5€ 🍷"
print(f"{text!r}: {len(text)} code points, {len(text.encode('utf-8'))} UTF-8 bytes, "
      f"{len(text.encode('utf-16-le'))} UTF-16 bytes")

# 3. Decode with the wrong encoding and you get mojibake; decode garbage and you get an error
raw = "café".encode("utf-8")
```

```

print(f"\n{raw!r} decoded as UTF-8: {raw.decode('utf-8')!r}; as Latin-1: {raw.decode('latin-1')!r}")
try: b"caf\xe9".decode("utf-8") # Latin-1 bytes read as UTF-8
except UnicodeDecodeError as e: print(f"b'caf\xe9'.decode('utf-8') -> UnicodeDecodeError: {e.reason}")
print(f"with errors='replace': {b'caf\xe9'.decode('utf-8', errors='replace')!r}")

# 4. One character on screen is not always one code point
composed, decomposed = "é", "ê" # é, or e followed by a combining accent
print(f"\n{composed!r} == {decomposed!r}: {composed == decomposed}; lengths {len(composed)} and {len(decomposed)}; after NFC normalisation equal: "
      f"{unicodedata.normalize('NFC', decomposed) == composed}")
for s in ("🤪", "🇯🇵"):
    print(f"{s} is {len(s)} code points: {' + '.join(f'U+{ord(c):04X}' for c in s)}")

# 5. Strings never change; operations return new strings
s = " Hello, World "
print(f"\n{s.strip()!r} {s.lower()!r} {s.split(',')!r} {'-'.join(['a', 'b', 'c'])!r}")
print(f"'Straße'.upper() = {'Straße'.upper()!r}; 'Straße'.casefold() == 'STRASSE'.casefold(): "
      f"'{Straße}'.casefold() == 'STRASSE'.casefold()}")
parts = []
for i in range(5): parts.append(str(i)) # build pieces, join once
print(f"''.join(parts) = {''.join(parts)!r}; f-string formatting: {3.14159:.2f} {42:>6} {1234567:,.3f}")

```

OUTPUT

```

'A': code point U+0041 (65), name LATIN CAPITAL LETTER A
'é': code point U+00E9 (233), name LATIN SMALL LETTER E WITH ACUTE
'€': code point U+20AC (8,364), name EURO SIGN
'🤪': code point U+1F600 (128,512), name GRINNING FACE

'A': 1 byte 01000001 same as Python: True
'é': 2 bytes 11000011 10101001 same as Python: True
'€': 3 bytes 11100010 10000010 10101100 same as Python: True
'🤪': 4 bytes 11110000 10011111 10011000 10000000 same as Python: True
'naïve café, 5€ 🤪': 16 code points, 23 UTF-8 bytes, 34 UTF-16 bytes

b'caf\xc3\xa9' decoded as UTF-8: 'café'; as Latin-1: 'cafÃ©'
b'caf\xe9'.decode('utf-8') -> UnicodeDecodeError: unexpected end of data
with errors='replace': 'caf '

'é' == 'é': False; lengths 1 and 2; after NFC normalisation equal: True
🤪 is 3 code points: U+1F469 + U+200D + U+1F4BB
🇯🇵 is 2 code points: U+1F1EF + U+1F1F5

'Hello, World' ' hello, world ' [' Hello', ' World '] 'a-b-c'
'Straße'.upper() = 'STRASSE'; 'Straße'.casefold() == 'STRASSE'.casefold(): True
''.join(parts) = '01234'; f-string formatting: 3.14 42 1,234,567

```

Decoding: the boundary where text goes wrong

Bytes become text only when decoded, and decoding needs the right encoding. Decode UTF-8 bytes as Latin-1 and each byte becomes its own character, so the two bytes of `é` turn into `Ã©`: this is **mojibake**, and its look, capital A with a tilde before an accented letter, is the signature of UTF-8 read as Latin-1 or Windows-1252. Decoding in the other direction usually fails outright, because Latin-1's single byte for `é` isn't a valid UTF-8 sequence, and Python raises `UnicodeDecodeError`. `errors="replace"` substitutes U+FFFD, the replacement character `🔍`, so a program can carry on and show where the damage is.

The rule that prevents nearly all of this: decode bytes to text as soon as they enter your program, work with `str` throughout, and encode only on the way out, naming the encoding every time. `open(path, encoding="utf-8")` is that rule for files. Without `encoding=`, Python uses the platform's default, which on many Windows machines isn't UTF-8, so a script that works on a Mac can fail on a colleague's PC. PEP 686 makes UTF-8 Python's default everywhere from version 3.15, but writing `encoding="utf-8"` costs nothing and works on every version.

What a "character" is

`len` counts code points, and a character on screen can be several. `é` can be the single code point U+00E9 or the letter `e` followed by U+0301, a combining accent; they look identical, but they aren't equal and have different lengths. Text from different sources mixes the two forms, so before comparing or using text as a key, **normalise** it: `unicodedata.normalize("NFC", s)` converts to the composed form wherever possible, and then the example's two `é`s compare equal.

Some characters are always several code points. The emoji for a technologist is a woman, a zero-width joiner and a laptop, three code points drawn as one picture. A flag is two "regional indicator" letters. What a reader sees as one character is called a **grapheme cluster**, and slicing a string by code points can cut one in half. If you need to truncate user-visible text or count what a person would count, use a library that understands grapheme clusters, such as the third-party `regex` module's `\X` pattern.

Strings in Python can't be changed; every method returns a new string. That's why building a long string with `+=` in a loop can be slow, since each step may copy everything so far, and why the idiom is to collect pieces in a list and `"".join(pieces)` once at the end. Case is subtler than it looks too: `"Straße".upper()` is `"STRASSE"`, longer than the original, and for case-insensitive comparison `casefold()` is the correct method, since `lower()` leaves `ß` as it is.

COMMON TRAPS

Opening files without `encoding="utf-8"`, so the result depends on the machine.

Mixing `bytes` and `str` deep inside a program instead of decoding at the edge.

Comparing text from different sources without normalising it first.

Truncating to a byte or code-point length and cutting a character or an emoji in half.

Using `lower()` for case-insensitive matching instead of `casefold()`.

EXERCISE 8.1 Counting bytes by hand

Without running code, work out how many bytes `"naïve café, 5€ 😊"` takes in UTF-8, and check your answer against the example. How many would it take in UTF-16, and why is that 34 and not 32?

SOLUTION

The string has 16 code points. Four aren't ASCII: `ï` and `é` are below U+0800 and take 2 bytes each, `€` is below U+10000 and takes 3, and `😊` takes 4. The other 12 are ASCII at 1 byte each. Total: $12 + 2 + 2 + 3 + 4 = 23$, which matches. UTF-16 uses 2 bytes for every code point up to U+FFFF, which would give 32 for 16 code points, but code points above U+FFFF, like the emoji, don't fit in 16 bits and take two 16-bit units, a **surrogate pair**: so $15 \times 2 + 4 = 34$. UTF-16 is what Windows, Java and JavaScript use internally, which is why `"😊".length` is 2 in JavaScript but `len("😊")` is 1 in Python.

EXERCISE 8.2 Repairing mojibake

A database contains `"cafÃ©"` where `"café"` was meant. Undo the damage in one expression, and explain why the repair works. When would it fail?

SOLUTION

`"cafÃ©".encode("latin-1").decode("utf-8")` gives `"café"`. The damage was UTF-8 bytes decoded as Latin-1, which maps each byte to the character with the same number. Encoding as Latin-1 reverses that exactly, recovering the original bytes `63 61 66 C3 A9`, and decoding those correctly as UTF-8 gives the intended text. It fails when the wrong decoding wasn't Latin-1 but Windows-1252, which maps a few byte values to different characters, such as 0x80 to `€`; then use `encode("cp1252")`. It also fails if the mojibake was itself re-encoded more than once, or if some bytes were lost or replaced with U+FFFD along the way, after which the information is gone. The library `ftfy` automates these repairs for real data. Prevention is still much easier: name the encoding at every boundary.

EXERCISE 8.3 Truncating safely

A database column holds at most 10 bytes of UTF-8. Write `fit(text, limit)` that returns the longest prefix of `text` whose encoding fits, without producing invalid UTF-8. Test it on `"Crème brûlée"`.

SOLUTION

```
def fit(text, limit):
    data = text.encode("utf-8")[:limit]
    return data.decode("utf-8", errors="ignore")

assert fit("Crème brûlée", 10) == "Crème br"
assert len(fit("Crème brûlée", 10).encode("utf-8")) <= 10
```

Cutting the bytes at 10 may split a multi-byte character, here the `û`, whose first byte would be byte 10. Decoding with `errors="ignore"` drops the incomplete sequence at the end, which is exactly the partial character, because UTF-8's header bits make an incomplete sequence detectable. The result is "Crème br", 9 bytes. This keeps code points whole but not grapheme clusters: an `e` with a separate combining accent could lose its accent, and an emoji sequence could lose part of itself. For text shown to people, truncate by grapheme clusters, and only then check the byte length.

CHECKLIST

- ☐ Can you explain the difference between code points, characters on screen, and bytes?
- ☐ Can you encode a code point in UTF-8 by hand and say how many bytes it needs?
- ☐ Can you recognise mojibake and explain which wrong decoding produced it?
- ☐ Can you apply the rule: decode at the edge, work in `str`, encode at the edge, always naming the encoding?
- ☐ Can you explain normalisation and grapheme clusters, and when `len` misleads?

GO DEEPER

DOCS [Unicode HOWTO Python documentation](#)

ARTICLE [The Absolute Minimum Every Software Developer Must Know About Unicode Joel Spolsky](#)

TALK [Pragmatic Unicode, or, How do I stop the pain? Ned Batchelder](#)

Files, CSV and JSON

3 exercises

Real data arrives in files, and real files are messy: names with stray spaces, two date formats in one column, a pound sign in a price, a quantity typed as a word, a blank line in the middle. This chapter reads such a file, turns each row into a clean record or rejects it with a reason, totals the results, and writes them out as CSV and JSON. On the way it covers opening and closing files safely, why the `csv` module exists, what JSON can and can't represent, and processing a file too big to load. It's a small version of one of the stage's four deliverables.

Opening and closing files

`open(path, encoding="utf-8")` returns a file object, which reads text; mode `"w"` writes, replacing what was there, `"a"` appends, and adding `"b"` reads or writes bytes instead of text. A file must be closed when you're done, which flushes anything still buffered and releases the operating system's handle. The `with` statement does that for you: `with open(...) as f:` closes the file when the block ends, whether it ends normally or because of an exception, and it's the only way you should open files. The same pattern manages other resources, and the example uses it for a temporary folder that's deleted, contents and all, when its block ends.

The `pathlib` module makes paths objects rather than strings. `Path(folder) / "sales.csv"` joins paths with the right separator for the operating system, `path.read_text(encoding="utf-8")` and `write_text` read or write a whole small file in one call, `path.exists()` and `path.suffix` answer questions, and `folder.glob("*.csv")` finds files. It's clearer than building paths by joining strings, and it avoids the classic bug of a missing or doubled slash.

Why CSV needs a module

A CSV file looks simple enough to parse with `line.split(",")`, and that works until a field contains a comma. Then the field is quoted, `"Smith, Jo"`, a quote inside a quoted field is doubled, and a quoted field may even contain a newline. The `csv` module handles all of it. `csv.DictReader` uses the first row as column names and yields each row as a dict, so code says `row["Price"]` instead of `row[4]`, and it skips blank lines. When opening a file for `csv`, pass `newline=""`, so that the module, not Python's line-ending translation, decides where records end. `csv.writer` quotes whatever needs quoting on the way out, as the report's `"Smith, Jo"` shows.

EXAMPLE

files.py · a messy CSV cleaned row by row with reasons for every rejection, a report written as CSV and JSON, and a file read line by line

```
import csv, json, pathlib, tempfile
from datetime import date, datetime
from collections import defaultdict

MESSY = """Date,Customer,Product,Qty,Price
2026-01-03, Ana Lima ,Widget,2,"9.50"
03/01/2026,ana lima,WIDGET,1,9.5
2026-01-04,"Smith, Jo",Gadget,1,25.00
2026-01-05,Ben Ode,Gizmo,3,£12.00

2026-01-06,Ben Ode,Gizmo,two,12.00
2026-13-07,Cy Ito,Widget,4,9.50
2026-01-08,Cy Ito,Widget,4,9.50
2026-01-09,Ben Ode,Gadget,,25.00
"""

def parse_date(text):
    for fmt in ("%Y-%m-%d", "%d/%m/%Y"):
        try: return datetime.strptime(text.strip(), fmt).date()
        except ValueError: pass
    raise ValueError(f"unrecognised date {text!r}")

def clean(row):
    """Turn one raw CSV row into a tidy record, or raise ValueError saying why it can't be."""
    if not row["Qty"].strip(): raise ValueError("missing quantity")
    return {"date": parse_date(row["Date"]),
            "customer": " ".join(row["Customer"].split()).title(), # trim, squeeze spaces,
                        Title Case
            "product": row["Product"].strip().capitalize(),
            "qty": int(row["Qty"]), # 'two' raises ValueError
            "price": float(row["Price"].strip().lstrip("£$€"))}

with tempfile.TemporaryDirectory() as folder:
    src = pathlib.Path(folder) / "sales.csv"
    src.write_text(MESSY, encoding="utf-8") # always say which encoding

    good, rejected = [], []
    with src.open(newline="", encoding="utf-8") as f: # newline="" lets csv handle line
        endings
        reader = csv.DictReader(f) # skips blank lines by itself
        for row in reader:
            try: good.append(clean(row))
            except ValueError as e: rejected.append((reader.line_num, str(e)))

    print(f"{len(good)} clean rows, {len(rejected)} rejected:")
    for line_no, why in rejected: print(f"    line {line_no}: {why}")

    revenue = defaultdict(float)
    for r in good: revenue[r["customer"]] += r["qty"] * r["price"]
    print("\nrevenue by customer:")
    for customer, total in sorted(revenue.items(), key=lambda kv: -kv[1]):
        print(f"    {customer:<10} £{total:>7.2f}")

    # Write the report as CSV and as JSON, then read the JSON back
    out = pathlib.Path(folder) / "report.csv"
    with out.open("w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["customer", "revenue"])
        for customer, total in sorted(revenue.items()): writer.writerow([customer, f"
            {total:.2f}"])
```

```

print(f"\nreport.csv:\n{out.read_text(encoding='utf-8')}", end="")

record = {"rows": len(good), "first": good[0]["date"].isoformat(), "pair": (1, 2), 7:
"seven"}
text = json.dumps(record)
back = json.loads(text)
print(f"\nJSON text: {text}")
print(f"read back: {back}")
print("    the tuple came back as a list, and the key 7 as the string '7'")
try: json.dumps({"when": date(2026, 1, 3)})
except TypeError as e: print(f"a date isn't JSON: {e}; store date.isoformat() instead")

with src.open(encoding="utf-8") as f:                # a file is an iterator of lines: no
    need to
    widest = max(len(line.rstrip("\n")) for line in f) # load it all to process it
    print(f"\nthe longest line in sales.csv has {widest} characters")
print(f"after the with block, the temporary folder exists: {src.parent.exists()}")

```

OUTPUT

```

5 clean rows, 3 rejected:
  line 7: invalid literal for int() with base 10: 'two'
  line 8: unrecognised date '2026-13-07'
  line 10: missing quantity

revenue by customer:
  Cy Ito      £  38.00
  Ben Ode     £  36.00
  Ana Lima    £  28.50
  Smith, Jo   £  25.00

report.csv:
customer,revenue
Ana Lima,28.50
Ben Ode,36.00
Cy Ito,38.00
"Smith, Jo",25.00

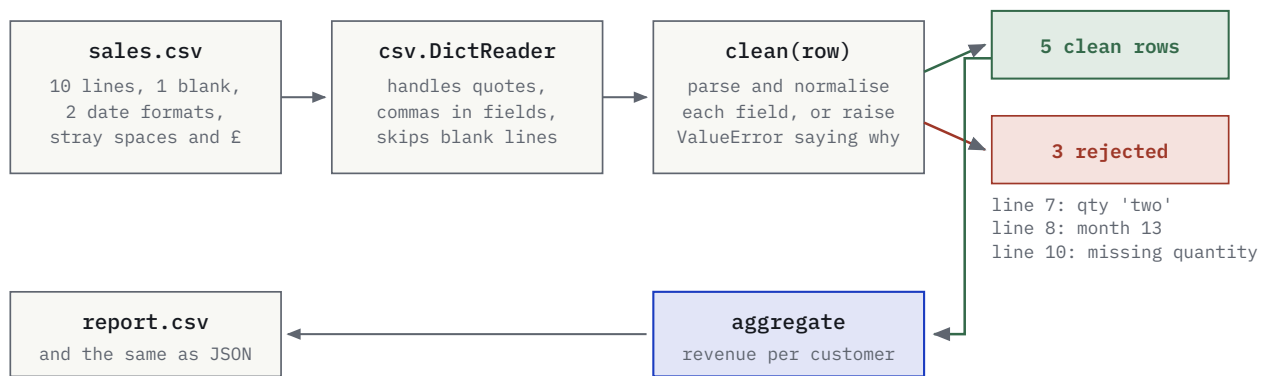
JSON text: {"rows": 5, "first": "2026-01-03", "pair": [1, 2], "7": "seven"}
read back: {'rows': 5, 'first': '2026-01-03', 'pair': [1, 2], '7': 'seven'}
    the tuple came back as a list, and the key 7 as the string '7'
a date isn't JSON: Object of type date is not JSON serializable; store date.isoformat() instead

the longest line in sales.csv has 37 characters
after the with block, the temporary folder exists: False

```

Cleaning: one clean record or one clear reason

The heart of the example is `clean(row)`, which either returns a tidy record or raises `ValueError` saying exactly what's wrong. It trims and squeezes spaces and fixes the capitalisation of names, so `" Ana Lima "` and `"ana lima"` become the same customer; strips currency symbols; parses the two date formats the file actually uses; and converts quantities and prices to numbers, which fails, correctly, for `"two"`. The loop around it collects the good records and, for every failure, the line number and the reason, using the reader's `line_num` rather than counting rows, since counting would be thrown off by the skipped blank line.



bad rows are reported with their line numbers, never silently dropped or guessed at

Every row ends in exactly one of two places: a clean record, or the rejection list with its line number and reason. The report is built only from the first.

Two principles make this robust. **Validate at the boundary:** the rest of the program receives only records that are already correct, so it never has to check for a price with a pound sign in it. And **never guess.** The row with month 13 could be a typo for the 3rd of July or the 7th of March; the program can't know, so it rejects the row and says why, and a person decides. Guessing silently produces a report that looks fine and is wrong. The date `03/01/2026` shows the subtler version of the problem: it's the 3rd of January in Britain and March 1st in the United States. The example reads it day-first, the British convention, because it's a British shop's file. That's a decision about this particular source, and it belongs in a comment and a test, not in the reader's head.

JSON

JSON is the standard format for structured data between programs and in most web APIs. It has objects, arrays, strings, numbers, `true`, `false` and `null`, which correspond to Python's dicts, lists, strings, numbers, `True`, `False` and `None`; `json.dumps` converts a value to JSON text and `json.loads` converts back. The correspondence isn't exact, and the example shows the losses: a tuple comes back as a list, a dict key 7 comes back as the string `"7"`, because JSON keys are always strings, and a `date` can't be written at all. The fix is to convert such values yourself, a date to `date.isoformat()`, and back after loading. `json.dumps(data, indent=2, sort_keys=True)` produces readable, stable output for files that people will read or compare in version control.

Python's `pickle` module can save almost any Python object, which makes it tempting. Don't use it for data that comes from outside your program: loading a pickle can run arbitrary code, so a malicious file can take over your machine. JSON, CSV and other plain formats can only describe data.

Files that don't fit in memory

A file object is an **iterator** over its lines: `for line in f` reads one line at a time, using memory for one line however big the file is. `f.read()` and `read_text` load everything at once, which is fine for small files and fails for a 50-gigabyte log. The example finds the longest line of the file by streaming it through `max`. The same applies to CSV: `DictReader` reads lazily, so a loop that updates running totals per customer needs memory proportional to the number of customers, not the number of rows.

COMMON TRAPS

Splitting CSV lines on commas by hand, which breaks on the first quoted field.

Opening files without `with`, so an exception leaves them open and unflushed.

Silently skipping or "fixing" bad rows, so the report is wrong and nobody knows.

Assuming JSON round-trips Python values, and finding tuples turned to lists and int keys to strings.

Unpickling data from anywhere you don't control.

EXERCISES

EXERCISE 9.1 An errors file

Change the example to write the rejected rows to `errors.csv`, with the original row, its line number and the reason. Why is that more useful than printing them?

SOLUTION

```
rejected.append((reader.line_num, str(e), row))      # keep the row as well
...
with (pathlib.Path(folder) / "errors.csv").open("w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["line", "reason", *reader.fieldnames])
    for line_no, why, row in rejected:
        writer.writerow([line_no, why, *row.values()])
```

Printed messages scroll away; a file can be handed to the person who owns the data, opened in a spreadsheet, fixed and fed back in. Including the original row, not just its line number, lets them correct it without opening the source file, and including the reason tells them what to correct. It also makes the program's behaviour checkable: the

number of clean rows plus the number of error rows must equal the number of data rows, an invariant worth asserting at the end of every cleaning job, since it proves no row was lost.

EXERCISE 9.2 Ambiguous dates

A new supplier's file uses slashes for every date, and you don't know whether it's day-first or month-first. Describe how a program could decide from the data itself, and what it should do when it can't.

SOLUTION

Parse every date both ways. Any value whose first number is greater than 12, such as `25/03/2026`, can only be day-first, and any whose second number is greater than 12 can only be month-first. If the file contains only the first kind of evidence, it's day-first; only the second, month-first; both, the file is inconsistent and should be rejected as a whole. If neither appears, because every day happens to be 12 or below, the data can't decide, and the program must not guess: stop with a message asking for the format, and accept it as a setting for that supplier from then on. A weaker hint is order: a file sorted by date is only sorted under one interpretation. The general lesson is that formats must be stated, not inferred, which is why ISO 8601, `2026-01-03`, exists: it sorts correctly as text and can't be misread.

EXERCISE 9.3 A ten-gigabyte file

The same sales format arrives as a 10 GB file with 100 million rows and 50,000 customers. Explain why the example's approach still works, which parts use memory in proportion to what, and what one change would be needed.

SOLUTION

`DictReader` reads the file one record at a time, and `clean` handles one row at a time, so neither depends on the file's size. The `revenue` dict holds one total per customer, 50,000 entries, a few megabytes. The one problem is `good`, which keeps every clean record in a list: 100 million dicts would need tens of gigabytes. The fix is to update the totals as each row is cleaned, without storing the row, `revenue[r["customer"]]` `+= r["qty"] * r["price"]` inside the loop, and to write rejected rows to the errors file as they occur rather than collecting them. Then memory is proportional to the number of customers, and the running time to the number of rows, which is the best possible for a job that must look at every row once.

CHECKLIST

- ☐ Can you open files with `with` and an explicit encoding, and use `pathlib` for paths?
- ☐ Can you read and write CSV with the `csv` module, including quoted fields?
- ☐ Can you write a cleaning function that returns a clean record or raises with a reason?
- ☐ Can you say which Python values JSON preserves, which it changes, and which it can't store?
- ☐ Can you process a file larger than memory by streaming it?

GO DEEPER

DOCS [The Python Tutorial: Input and Output Python documentation](#)

DOCS [csv: CSV File Reading and Writing Python documentation](#)

ARTICLE [Use international date format \(ISO\) W3C](#)

PART III

Getting it right

Three chapters on the difference between a program that works on the input you tried and one that works. Errors and exceptions are how a program reports that something went wrong, and the skill is deciding where to detect a problem and where to handle it. Tests are how you find out that something went wrong before a user does, and they belong in a project from its first week. A debugger is how you find out why, by watching the program run instead of guessing. Together they replace "it seems fine" with evidence.

Errors and exceptions

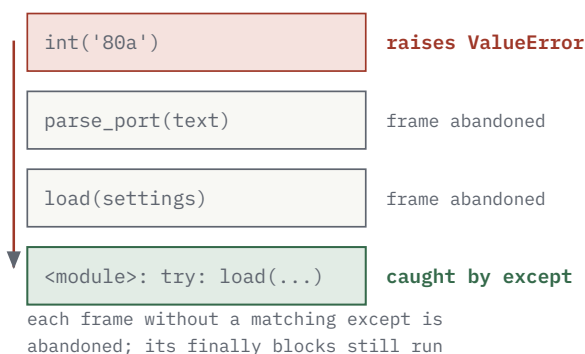
3 exercises

When something goes wrong, a Python operation raises an exception: it stops what it's doing and hands the problem to whoever called it, and so on up the call stack, until some code catches it or the program ends with a traceback. Used well, exceptions put each problem where it can be handled, and make bugs loud instead of silent. Used badly, with a catch-everything `except` or a function that returns `None` when it fails, they hide bugs until they surface somewhere unrelated. This chapter covers raising, catching and chaining, and the design question that matters most: at which boundary to fail.

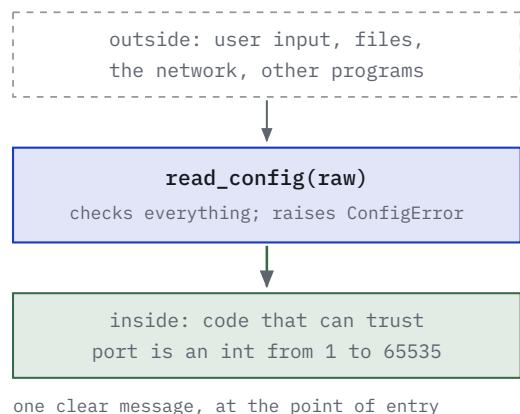
What happens when an exception is raised

`int("80a")` can't produce a number, so it raises a `ValueError`. Execution doesn't continue to the next line. Python looks for a handler in the current function, a `try` block with a matching `except`; if there isn't one, the function's frame is abandoned and Python looks in its caller, and so on up the stack. The first matching `except` runs, and execution continues after its `try` statement. If nothing matches, the program stops and prints a **traceback**: the chain of calls, outermost first, and on the last line the exception's type and message.

AN EXCEPTION UNWINDS THE STACK



VALIDATE AT THE BOUNDARY



Two pictures of where errors go. An exception travels up the call stack to the nearest handler that matches; a good design checks data once, where it enters, so the code inside never has to.

Read tracebacks from the bottom. The last line says what went wrong, `ValueError: invalid literal for int() with base 10: '80a'`, and the lines above it say where, innermost call last. The bug is usually in the deepest frame that's your own code rather

than a library's, and often in what was passed to it. Python 3.11 and later also underline the exact expression on each line that raised, which settles questions like which of three indexes on one line was out of range.

EXAMPLE

errors.py · a traceback's path, catching narrowly, the order of try, except, else and finally, custom exceptions raised at the boundary, and the cost of returning None

```
import traceback

# 1. An exception unwinds the call stack until something catches it. The traceback is the path.
def load(settings): return parse_port(settings["port"])
def parse_port(text): return int(text)
try: load({"port": "80a"})
except ValueError as e:
    frames = traceback.extract_tb(e.__traceback__)
    print(f'{type(e).__name__}: {e}')
    print("    raised through: " + " -> ".join(f'{fr.name} (line {fr.lineno})' for fr in
        frames))

# 2. Catch narrowly. A bare except swallows the bug you didn't know you had.
def average_bad(xs):
    try: return sum(xs) / len(xss)           # a typo: xss
    except: return 0                        # hides the NameError, and every other bug
def average(xs):
    try: return sum(xs) / len(xss)
    except ZeroDivisionError: return 0       # only the case we meant to handle
print(f"\naverage_bad([2, 4]) = {average_bad([2, 4])} (wrong, and silent)")
try: average([2, 4])
except NameError as e: print(f"average([2, 4]) -> NameError: {e} (loud, and fixable)")

# 3. The order: try, then except if it failed, else if it didn't, finally always
def attempt(x):
    steps = []
    try:
        steps.append("try"); 10 / x
    except ZeroDivisionError:
        steps.append("except")
    else:
        steps.append("else")
    finally:
        steps.append("finally")
    return steps
print(f"\nattempt(5): {attempt(5)}; attempt(0): {attempt(0)}")

# 4. Raise your own exceptions, with a message that says what and where, at the boundary
class ConfigError(Exception):
    """Something is wrong with the configuration a user gave us."""
def read_config(raw):
    port = raw.get("port")
    if port is None: raise ConfigError("missing 'port'")
    try: port = int(port)
    except ValueError as e: raise ConfigError(f"'port' must be a whole number, got {port!r}")
        from e
    if not 1 <= port <= 65535: raise ConfigError(f"'port' must be 1 to 65535, got {port}")
    return {"port": port, "host": raw.get("host", "localhost")}
for raw in ({"port": "8080"}, {}, {"port": "80a"}, {"port": "70000"}):
    try: print(f'{raw} -> {read_config(raw)}')
    except ConfigError as e:
        cause = f" (caused by {type(e.__cause__).__name__})" if e.__cause__ else ""
        print(f'{raw} -> ConfigError: {e}{cause}')
```

```
# 5. Returning None for failure moves the crash somewhere else
def find_price_quiet(prices, item): return prices.get(item) # None if missing
prices = {"tea": 2.5, "cake": 3.0}
order = ["tea", "scone", "cake"]
try: total = sum(find_price_quiet(prices, item) for item in order)
except TypeError as e: print(f"\nquiet failure, found later: TypeError: {e}")
def find_price(prices, item):
    try: return prices[item]
    except KeyError: raise KeyError(f"no price for {item!r}") from None
try: total = sum(find_price(prices, item) for item in order)
except KeyError as e: print(f"loud failure, found at the cause: KeyError: {e.args[0]}")
```

OUTPUT

```
ValueError: invalid literal for int() with base 10: '80a'
  raised through: <module> (line 6) -> load (line 4) -> parse_port (line 5)

average_bad([2, 4]) = 0 (wrong, and silent)
average([2, 4]) -> NameError: name 'xss' is not defined (loud, and fixable)

attempt(5): ['try', 'else', 'finally']; attempt(0): ['try', 'except', 'finally']
{'port': '8080'} -> {'port': 8080, 'host': 'localhost'}
{} -> ConfigError: missing 'port'
{'port': '80a'} -> ConfigError: 'port' must be a whole number, got '80a' (caused by
ValueError)
{'port': 70000} -> ConfigError: 'port' must be 1 to 65535, got 70000

quiet failure, found later: TypeError: unsupported operand type(s) for +: 'float' and
'NoneType'
loud failure, found at the cause: KeyError: no price for 'scone'
```

Catching

`try` wraps code that might fail; `except SomeError as e:` handles that type of error and its subclasses; `else` runs only if nothing was raised; `finally` runs no matter what, even if the `try` block returned or the exception is still propagating, which makes it the place for cleanup. The example prints the order both ways. In practice, `with` blocks replace most `finally` clauses, since a file or lock used with `with` is released automatically.

Catch as narrowly as you can, in two senses. Name the specific exception type you expect, and put only the line that can raise it inside the `try`. The example's `average_bad` shows why: its bare `except:` was meant for an empty list, but it also caught the `NameError` from a typo, so the function returned 0 for a list that averages 3, with no sign of a problem. A bare `except:` even catches `KeyboardInterrupt`, so the program ignores Ctrl-C. The narrow version lets the typo crash immediately, with a message naming the misspelt variable. A crash you can see is a gift compared with a wrong answer you can't.

There is one place where catching broadly is right: the top of a long-running program, such as a web server's request loop or a batch job's loop over records, where one failure shouldn't stop everything. There, `except Exception as e:` logs the full traceback and moves on to the next item. Note `Exception`, not a bare `except`, so that Ctrl-C and `sys.exit` still work.

Raising, and the right boundary

Raise exceptions in your own code with `raise ValueError(...)` when a function is given something it can't work with. A good message says what was expected and shows what was received: `"'port' must be 1 to 65535, got 70000"` lets the reader fix the input without reading your code. Define your own exception class, a subclass of `Exception`, when callers will want to handle your errors separately from Python's, as the example's `ConfigError` lets a caller report configuration problems to the user while letting genuine bugs crash.

Where should errors be detected? At the **boundary**, where data enters your program from outside: user input, files, network responses, other people's code. Check everything there, convert it to the types and ranges the rest of the program expects, and raise one clear error if it doesn't fit. Code inside the boundary can then trust its data, which makes it simpler and faster, and an error message appears once, at the point where it can name the bad input, instead of as a mysterious failure three functions later. Chapter 9's `clean` function was a boundary; so is the example's `read_config`.

The opposite habit, returning `None` to signal failure, moves the error somewhere worse. The example's `find_price_quiet` returns `None` for a scone with no price, and the failure appears later, inside `sum`, as a `TypeError` about adding a float to `None`, which says nothing about scones. Raising `KeyError("no price for 'scone'")` at the lookup names the real problem. When "not found" is a normal outcome rather than an error, returning `None` is fine, as long as the function's name and docstring say so and every caller checks.

When you catch one exception and raise another, chain them with `raise ConfigError(...) from e`. The new exception keeps the original as its `__cause__`, and a traceback shows both, so the detail isn't lost. `from None` deliberately hides the original, for when it's just noise, as in the example's `find_price`.

Asking forgiveness or permission

Python style generally prefers EAFP, "easier to ask forgiveness than permission": try the operation and handle the exception, rather than LBYL, "look before you leap", checking first. The reason is that checks can be wrong or out of date. `if path.exists(): open(path)` fails anyway if the file is deleted between the check and the open, or if it exists but you lack permission to read it; `try: open(path) except FileNotFoundError:` handles exactly the case that happened. Use `assert` for a different purpose: to state things that must be true if your code is correct, such as a loop invariant. Assertions can be disabled with `python -O`, so they must never be used to validate input.

COMMON TRAPS

A bare `except:` or `except Exception: pass`, which hides every bug in the block.

A `try` block around many lines, so you can't tell which one failed.

Returning `None` or `-1` to mean "error" and leaving every caller to remember to check.

Error messages without the bad value, such as "invalid input".

Using `assert` to validate user input, which `python -O` removes.

EXERCISES

EXERCISE 10.1 Reading a traceback

A program fails with the traceback below. Which line do you read first, which function do you open, and what's your first hypothesis?

```
Traceback (most recent call last):
  File "report.py", line 41, in <module>
    main(sys.argv[1])
  File "report.py", line 35, in main
    rows = [parse(line) for line in f]
  File "report.py", line 12, in parse
    return {"name": parts[0], "total": float(parts[3])}
IndexError: list index out of range
```

SOLUTION

Read the last line first: an `IndexError`, a list was indexed past its end. The frame just above it says where: line 12 of `parse`, and of the two indexes on that line, `parts[3]` is the likely one, since a line with at least one part makes `parts[0]` safe. (Python 3.11 and later would underline `parts[3]` exactly.) So the hypothesis is that some line of the input has fewer than four fields: a blank line, a header, or a line with a missing comma. The next frame up confirms the input comes straight from the file, unfiltered. To check, print or break on the failing line when `len(parts) < 4`. The fix belongs in `parse`, which is the boundary: check the number of fields and raise `ValueError(f"expected 4 fields, got {len(parts)}: {line!r}")`, so the next failure names the offending line.

EXERCISE 10.2 A function that says what's wrong

Write `parse_age(text)` that returns an integer age from user input, raising `ValueError` with a specific message for each kind of bad input: empty, not a whole number, negative, and implausibly large.

SOLUTION

```
def parse_age(text):
    text = text.strip()
```

```

if not text:
    raise ValueError("age is empty")
try:
    age = int(text)
except ValueError:
    raise ValueError(f"age must be a whole number, got {text!r}") from None
if age < 0:
    raise ValueError(f"age can't be negative, got {age}")
if age > 150:
    raise ValueError(f"age {age} is not plausible")
return age

```

Each check names the problem and shows the value, so a form can display the message directly. The `try` wraps only the `int` call, the one line that can raise `ValueError` for a reason we want to rephrase, and `from None` hides Python's less friendly original message. Order matters: stripping first means "42" is accepted, and checking for empty before converting gives a better message than `int("")` would. Tests for this function are easy to write, one per message, and chapter 11 shows how.

EXERCISE 10.3 Look before you leap

Rewrite this in EAFP style, and give two situations in which the original fails although the check passed.

```

if os.path.exists(path):
    with open(path, encoding="utf-8") as f:
        settings = json.load(f)
else:
    settings = {}

```

SOLUTION

```

try:
    with open(path, encoding="utf-8") as f:
        settings = json.load(f)
except FileNotFoundError:
    settings = {}

```

The original fails if the file is deleted or renamed between `exists` and `open`, a **race condition** that's rare in testing and real in production, where other programs share the disk. It also crashes, as it should, when the path exists but is a folder, or isn't readable, or contains invalid JSON, but its structure suggests those were considered and they weren't. The EAFP version handles precisely the case it means to, a missing file means default settings, and lets every other problem raise its own specific exception. Whether to catch `json.JSONDecodeError`

too is a design decision: silently using defaults when the settings file is corrupt would hide a real problem from the user, so it's usually better to let it fail with a message naming the file.

CHECKLIST

- ☐ Can you read a traceback from the bottom up and find the frame to investigate?
- ☐ Can you write `try`, `except`, `else` and `finally`, and say when each runs?
- ☐ Can you catch narrowly, and say where catching broadly is appropriate?
- ☐ Can you raise exceptions with messages that name the problem and the bad value?
- ☐ Can you explain why validating at the boundary beats returning `None` deep inside?

GO DEEPER

DOCS [The Python Tutorial: Errors and Exceptions](#) Python documentation

DOCS [Built-in Exceptions, and their hierarchy](#) Python documentation

ARTICLE [Parse, don't validate](#) Alexis King

Testing

3 exercises

A test is a small program that runs part of your program and checks the result. A suite of them, run in seconds after every change, is what lets you change code without fear, because anything you break is reported at once, with its name. Tests belong in a project from the first week, not after it works, since writing them is also how you discover what "works" means. This chapter tests a pair of functions with a real test runner, finds three bugs, and then finds a fourth, which every unit test missed, by checking a property on two thousand random inputs.

A test is a claim you can run

Each test states one fact about the code: `format_duration(5405)` is `"1h 30m 5s"`. It has three parts, often called **arrange**, **act**, **assert**: set up the input, call the code, check the result. A test passes silently and fails loudly, naming itself and showing what it got and what it expected. Give each test a name that states the behaviour it checks, such as `test_zero_is_written_as_0s`, because when it fails months from now, the name is the first thing anyone reads.

Python's standard library includes `unittest`, which the example uses so that it runs without installing anything. Most Python projects use `pytest` instead, and you should too: it finds test files named `test_*.py` by itself, lets tests be plain functions using plain `assert`, shows a detailed diff when an assertion fails, and has features such as `@pytest.mark.parametrize` for running one test with many inputs. The `pytest` version of the example's first test is two lines:

```
def test_formats_hours_minutes_seconds():
    assert format_duration(5405) == "1h 30m 5s"
```

Running `pytest` in the project folder runs every test and reports a summary. Run it after every change, and before every commit. A test suite that takes more than a few seconds stops being run, so keep unit tests fast: no network, no sleeping, no large files.

EXAMPLE

testing.py · six unit tests run against a buggy and a fixed version, then a seeded round-trip property check that finds the bug the unit tests missed

```
import random, re, unittest

# The code under test: durations like "1h 30m 5s" to seconds and back. Version 1 has bugs.
def format_v1(seconds):
    h, rest = seconds // 3600 % 24, seconds % 3600          # a clock's hours, not a duration's
    m, s = divmod(rest, 60)
```

```

        return " ".join(f"{n}{unit}" for n, unit in ((h, "h"), (m, "m"), (s, "s")) if n)
def parse_v1(text):
    total = 0
    for number, unit in re.findall(r"(\d+)([hms])", text):
        total += int(number) * {"h": 3600, "m": 60, "s": 1}[unit]
    return total

def format_v2(seconds):
    if seconds < 0: raise ValueError("a duration can't be negative")
    h, rest = divmod(seconds, 3600)
    m, s = divmod(rest, 60)
    return " ".join(f"{n}{unit}" for n, unit in ((h, "h"), (m, "m"), (s, "s")) if n) or "0s"
def parse_v2(text):
    if not re.fullmatch(r"\s*(\d+[hms]\s*)+", text): raise ValueError(f"not a duration: {text!r}")
    return sum(int(n) * {"h": 3600, "m": 60, "s": 1}[u] for n, u in re.findall(r"(\d+)([hms])", text))

def make_tests(fmt, parse):
    class DurationTests(unittest.TestCase):
        def test_formats_hours_minutes_seconds(self): self.assertEqual(fmt(5405), "1h 30m 5s")
        def test_skips_zero_parts(self): self.assertEqual(fmt(3600), "1h")
        def test_zero_is_written_as_0s(self): self.assertEqual(fmt(0), "0s")
        def test_pareses_what_it_formats(self): self.assertEqual(parse("1h 30m 5s"), 5405)
        def test_rejects_garbage(self):
            with self.assertRaises(ValueError): parse("ninety minutes")
        def test_rejects_negative(self):
            with self.assertRaises(ValueError): fmt(-5)
    return DurationTests

class Recorder(unittest.TestResult):
    # collects results; no timings, so output is stable
    def __init__(self): super().__init__(); self.passed = []
    def addSuccess(self, test): self.passed.append(test)
def run(fmt, parse, label):
    result = Recorder()
    unittest.defaultTestLoader.loadTestsFromTestCase(make_tests(fmt, parse)).run(result)
    print(f"{label}: {len(result.passed)} passed, {len(result.failures)} failed")
    for test, tb in result.failures:
        why = next(line for line in tb.splitlines() if line.startswith("AssertionError"))
        print(f"    FAIL {test._testMethodName}: {why}")
run(format_v1, parse_v1, "version 1")
run(format_v2, parse_v2, "version 2")

# A property check: for many random inputs, parsing what we formatted gives back the input
def round_trip_failures(fmt, parse, trials=2000):
    rng = random.Random(7) # seeded, so a failure can be reproduced
    bad = [s for s in (rng.randrange(0, 200_000) for _ in range(trials)) if parse(fmt(s)) != s]
    edges = (0, 59, 60, 3599, 3600, 86_399, 86_400)
    bad += [s for s in edges if parse(fmt(s)) != s] # and the edges, always
    return sorted(bad), trials + len(edges)
print()
for label, fmt, parse in (("version 1", format_v1, parse_v1), ("version 2", format_v2, parse_v2)):
    bad, tried = round_trip_failures(fmt, parse)
    if not bad: print(f"round trip, {label}: all {tried:,} inputs survive")
    else: print(f"round trip, {label}: {len(bad):,} of {tried:,} inputs fail; the smallest is "
                f"{bad[0]:,}, formatted as {fmt(bad[0])!r}, parsed back as {parse(fmt(bad[0])):,}")

```

OUTPUT

```

version 1: 3 passed, 3 failed
FAIL test_rejects_garbage: AssertionError: ValueError not raised
FAIL test_rejects_negative: AssertionError: ValueError not raised
FAIL test_zero_is_written_as_0s: AssertionError: '' != '0s'

```

```
version 2: 6 passed, 0 failed
```

```
round trip, version 1: 1,087 of 2,007 inputs fail; the smallest is 86,400, formatted as '',  
    parsed back as 0  
round trip, version 2: all 2,007 inputs survive
```

What the tests found

Version 1 failed three of six tests. `format(0)` returned an empty string instead of `"0s"`, because every part was zero and the join had nothing to join; `parse` accepted `"ninety minutes"` and returned 0, because its regular expression simply found no matches; and `format(-5)` didn't complain at all. Each failure message names the test and shows the assertion, `' ' != '0s'`, so the fix is obvious. Version 2 handles all three, and all six tests pass.

Notice what kind of bugs these are. None is in the main path, formatting an ordinary duration like an hour and a half, which version 1 got right. They're all at **edges**: zero, invalid input, negative input. That's where bugs live, and so that's where tests should go. For any function, test the empty or zero case, a single item, a typical case, the boundaries of every range, and each kind of invalid input, which should raise an error. `assertRaises` in `unittest`, and `with pytest.raises(ValueError):` in `pytest`, check that an error is raised when it should be.

The bug the unit tests missed

Version 1 has a fourth bug, and all six unit tests passed it. Its hours are computed as `seconds // 3600 % 24`, the hours of a clock rather than of a duration, so anything longer than a day loses whole days. Every unit test used durations under a day, so none could see it. A different style of test can. A **property** is a statement that should hold for every input: here, "parsing what we formatted gives back the original number". The example checks it on 2,000 random durations up to 200,000 seconds, plus a list of edge values, and 1,087 of version 1's inputs fail. The smallest failure, 86,400 seconds, exactly one day, formats as an empty string.

WHERE THE TESTS LOOKED, ON THE INPUTS 0 TO 200,000 SECONDS



unit tests: 0, 3,600 and 5,405 seconds, all inside the first day

random round-trip checks: spread over the whole range, so most of them land in the bug

Why the property check found what the unit tests didn't. Hand-picked inputs cluster where the author was thinking; random inputs go everywhere, including the region the author never considered.

Two details make property checks practical. The random generator is **seeded**, `random.Random(7)`, so every run checks the same inputs and a failure can be reproduced exactly; an unseeded test that fails once and then passes is worse than none. And reporting the *smallest* failing input makes the failure easy to understand, since 86,400 says "one day" in a way that 173,962 doesn't. The **Hypothesis** library automates both, generating inputs of any type and shrinking every failure to a minimal example; it's well worth learning once you're comfortable with ordinary tests.

Habits that make tests worth having

Watch every new test **fail** before you make it pass. A test that has never failed might not be testing anything, if it calls the wrong function, say, or asserts something always true. Writing the test first and then the code, test-driven development, is one way to guarantee that, and it makes you decide what the function should do before deciding how. When you fix a bug, first write the test that reproduces it, a **regression test**, so the bug can never quietly come back.

Keep tests independent: each sets up what it needs, so they pass in any order and one failure doesn't cascade. Keep them deterministic: code that reads the clock, the network or an unseeded random generator should take those as arguments so a test can supply fixed ones, a design chapter 20 uses throughout. And treat coverage as a floor, not a goal. The `coverage` tool shows which lines your tests never ran, which is useful; but version 1's unit tests ran every one of its lines, and it was still wrong for every input over a day.

COMMON TRAPS

Testing only the typical case, when bugs live at zero, empty, one, boundaries and invalid input.

Writing tests after the code is "finished", when they confirm what the code does rather than what it should do.

Tests that depend on the clock, the network or unseeded randomness, which fail sometimes and teach people to ignore failures.

Never seeing a test fail, so you don't know whether it can.

Reading 100% line coverage as "correct". Every line ran in version 1.

EXERCISES

EXERCISE 11.1 Tests for `parse_age`

Write pytest tests for chapter 10's `parse_age`, using `parametrize` for the valid inputs and for the invalid ones.

SOLUTION

```
import pytest
from ages import parse_age

@pytest.mark.parametrize("text, age", [
    ("42", 42), (" 7 ", 7), ("0", 0), ("150", 150)])
def test_accepts_valid_ages(text, age):
    assert parse_age(text) == age

@pytest.mark.parametrize("text, message", [
    ("", "empty"), ("forty", "whole number"), ("4.5", "whole number"),
    ("-1", "negative"), ("151", "plausible")])
def test_rejects_invalid_ages(text, message):
    with pytest.raises(ValueError, match=message):
        parse_age(text)
```

Each row of a parametrised test runs and reports as a separate test, so one failure doesn't hide the others. The valid cases include both boundaries, 0 and 150, and input with spaces. The invalid cases cover each rule and check the message with `match`, which is a regular expression, so that a function raising the right type of error for the wrong reason still fails. Testing at 150 and 151 pins down the exact boundary, so changing `> 150` to `>= 150` by accident would be caught.

EXERCISE 11.2 The missing unit test

Without random testing, what single unit test would have caught version 1's day bug? What general rule would have led you to write it?

SOLUTION

`assert format_duration(90_000) == "25h"`, or any duration of a day or more: version 1 returns `"1h"` for 90,000 seconds. Better still, a pair at the boundary, 86,399 and 86,400 seconds, which version 1 formats as `"23h 59m 59s"` and an empty string. The rule is to look at every constant and every unit conversion in the code and test both sides of it. The code divides by 3,600 and 60, so durations around an hour and a minute deserve tests; and the moment you ask "what's the largest duration we'll ever format?", hours beyond 24 come up. Random testing finds such cases without anyone thinking of them, which is its value, but it doesn't replace thinking about boundaries; the two are complements.

EXERCISE 11.3 Testing code that reads the clock

This function can't be tested reliably. Change it so it can, and write the test.

```
def greeting(name):
    hour = datetime.now().hour
    return f"Good {'morning' if hour < 12 else 'afternoon'}, {name}"
```

SOLUTION

```
from datetime import datetime

def greeting(name, now=None):
    now = datetime.now() if now is None else now
    return f"Good {'morning' if now.hour < 12 else 'afternoon'}, {name}"

def test_greeting_changes_at_noon():
    assert greeting("Ana", datetime(2026, 1, 5, 11, 59)) == "Good morning, Ana"
    assert greeting("Ana", datetime(2026, 1, 5, 12, 0)) == "Good afternoon, Ana"
```

The original's output depends on when it runs, so a test of it passes in the morning and fails after lunch. Making the time a parameter, with the real clock as the default, changes nothing for callers but lets a test pass any moment it likes. The test checks both sides of the boundary at noon, which a test run at a random time of day would almost never do. The same technique, passing in what the function would otherwise reach out and get, works for random number generators, the network, files and environment variables, and it's called **dependency injection**.

CHECKLIST

- ☐ Can you write tests with pytest, including parametrised tests and tests that expect an exception?
- ☐ Can you list the edge cases worth testing for a given function?
- ☐ Can you write a property check with a seeded generator and report the smallest failure?
- ☐ Can you make code that uses the clock or randomness testable by passing them in?
- ☐ Can you explain why a test should fail before it passes, and why coverage isn't correctness?

GO DEEPER

DOCS [pytest: Get Started](#) [pytest developers](#)

DOCS [Hypothesis: property-based testing for Python](#) [Hypothesis developers](#)

BOOK [Test-Driven Development with Python](#) [Harry Percival](#)

Debugging

3 exercises

Debugging is finding the difference between what you think the program does and what it does. Guessing and changing things until the symptom goes away is the slow way, and it often leaves the bug in place. The fast way is to look: stop the program where the state goes wrong, inspect the variables, and compare them with what you expected. This chapter runs a real debugger session against a buggy function, shows what stepping through code reveals, and ends with bisection, the technique for when you don't even know where to look.

A method, not a mood

Every debugging session follows the same loop. **Reproduce** the failure reliably, ideally with the smallest input that shows it; a bug you can't reproduce, you can't confirm you've fixed. **Observe** what actually happens, rather than reasoning about what should. Form a **hypothesis** about the cause that would explain everything you've observed, and **test** it with one experiment, changing one thing at a time. When the hypothesis survives, fix the cause, not the symptom, then confirm the original failure is gone and write the test that would have caught it, as chapter 11 recommends.

The commonest mistake is skipping straight to the fix. A function returns 1.5 where 3 was expected; it's tempting to "fix" it by multiplying by 2, and the code is now wrong in two places. The example's `mean_of_evens([1, 2, 3, 4])` is that situation: the total looks right, so the question is what it's being divided by.

The debugger

A **debugger** runs your program under your control. You set a **breakpoint** on a line, and when execution reaches it the program pauses, with every variable available to inspect. Python's built-in debugger is `pdb`. Put `breakpoint()` on any line of your code, run the program normally, and you get a `(Pdb)` prompt there; or start the whole program under it with `python -m pdb script.py`. Editors such as VS Code and PyCharm put the same operations behind buttons and show the variables in a panel, and everything below applies to them too.

A handful of commands do almost all the work. `p expr` prints the value of any expression, and `pp` pretty-prints large ones. `n`, next, runs the current line and stops at the next one; `s`, step, does the same but goes inside any function the line calls. `c`, continue, runs until the next breakpoint. `b 9` sets a breakpoint at line 9, and `b 9, n % 2 == 1` sets a **conditional breakpoint** that only stops when the condition is true, which is indispensable

in a loop that runs a million times and goes wrong once. `w`, where, shows the call stack, and `u` and `d` move up and down it to inspect callers' variables. `l` lists the source around the current line, and `q` quits.

EXAMPLE

debugging.py · two scripted pdb sessions on a buggy function, a line-by-line trace of what stepping shows, and bisection to find which version broke

```
import bdb, io, linecache, os, pdb, sys

def mean_of_evens(nums):
    total = 0
    count = 0
    for n in nums:
        if n % 2 == 0:
            total += n
            count += 1
    return total / count
print(f"mean_of_evens([1, 2, 3, 4]) = {mean_of_evens([1, 2, 3, 4])}, but the evens 2 and 4 average 3")

# 1. A debugger session, scripted here so you can see it: the same commands work typed by hand
class Session(pdb.Pdb):
    def precmd(self, line):
        # echo each command after the prompt, as a terminal
        # would
        if not line.startswith("_pdbcmd"): self.stdout.write(line + "\n")
        return line
def debug(commands, func, *args):
    out = io.StringIO()
    session = Session(stdin=io.StringIO(commands), stdout=out, nosigint=True)
    session.use_rawinput = False
    session.runcall(func, *args)
    session.clear_all_breaks()
    bdb.Breakpoint.next, bdb.Breakpoint.bplist, bdb.Breakpoint.bpbynumber = 1, {}, [None]
    text, lines, skip = out.getvalue(), [], False
    for path in {os.path.realpath(__file__), os.path.abspath(__file__)}:
        # print a short name
        text = text.replace(path, "debugging.py")
    for line in text.splitlines():
        if "bdb.py" in line: skip = True; continue
        if skip: skip = False; continue
        lines.append(" " + line)
    print("\n".join(lines))
print("\nstop at the return and look at the variables:")
debug("break 10\ncontinue\np total, count\nwhere\ncontinue\n", mean_of_evens, [1, 2, 3, 4])
print("\nstop inside the loop only when n is odd, and watch count:")
debug("break 9, n % 2 == 1\ncontinue\np n, total, count\ncontinue\np n, total, count\n"
      "clear 1\ncontinue\n", mean_of_evens, [1, 2, 3, 4])

# 2. What stepping shows you, as a trace: each line as it runs, and the variables it changed
def trace_lines(func, *args):
    state = {"line": None, "seen": {}}
    def report(frame):
        changed = {k: v for k, v in frame.f_locals.items()
                    if k != "nums" and state["seen"].get(k, object()) != v}
        state["seen"] = dict(frame.f_locals)
        text = linecache.getline(frame.f_code.co_filename, state["line"]).split("#")[0].strip()
        print(f" {text:<22} {'', '.join(f' {k} = {v!r}' for k, v in
            changed.items())}").rstrip()
    def tracer(frame, event, arg):
        if frame.f_code is not func.__code__: return tracer
        if event in ("line", "return") and state["line"]: report(frame)
        # the previous line
        # is done
        state["line"] = None
```

```

        if event == "line": state["line"] = frame.f_lineno
        return tracer
    sys.settrace(tracer)
    try: return func(*args)
    finally: sys.settrace(None)
print("\nstepping through mean_of_evens([1, 2]), line by line:")
trace_lines(mean_of_evens, [1, 2])

# 3. When you don't know where the bug is, but you know when it wasn't there: bisect
versions = list(range(1, 65))          # 64 versions; the bug arrived in version 42
def broken(version): return version >= 42
lo, hi, checks = 0, len(versions) - 1, 0    # versions[lo] is good, versions[hi] is bad
while hi - lo > 1:
    mid = (lo + hi) // 2; checks += 1
    if broken(versions[mid]): hi = mid
    else: lo = mid
print(f"\nbisecting 64 versions: version {versions[hi]} introduced the bug, found in {checks}
      checks")

```

OUTPUT

mean_of_evens([1, 2, 3, 4]) = 1.5, but the evens 2 and 4 average 3

stop at the return and look at the variables:

```

> debugging.py(4)mean_of_evens()
-> total = 0
(Pdb) break 10
Breakpoint 1 at debugging.py:10
(Pdb) continue
> debugging.py(10)mean_of_evens()
-> return total / count
(Pdb) p total, count
(6, 4)
(Pdb) where
> debugging.py(10)mean_of_evens()
-> return total / count
(Pdb) continue

```

stop inside the loop only when n is odd, and watch count:

```

> debugging.py(4)mean_of_evens()
-> total = 0
(Pdb) break 9, n % 2 == 1
Breakpoint 1 at debugging.py:9
(Pdb) continue
> debugging.py(9)mean_of_evens()
-> count += 1          # the bug: this counts every number, not just evens
(Pdb) p n, total, count
(1, 0, 0)
(Pdb) continue
> debugging.py(9)mean_of_evens()
-> count += 1          # the bug: this counts every number, not just evens
(Pdb) p n, total, count
(3, 2, 2)
(Pdb) clear 1
Deleted breakpoint 1 at debugging.py:9
(Pdb) continue

```

stepping through mean_of_evens([1, 2]), line by line:

total = 0	total = 0
count = 0	count = 0
for n in nums:	n = 1
if n % 2 == 0:	
count += 1	count = 1
for n in nums:	n = 2
if n % 2 == 0:	
total += n	total = 2

```
count += 1          count = 2
for n in nums:
    return total / count
```

bisecting 64 versions: version 42 introduced the bug, found in 6 checks

What the sessions show

The first session stops at the `return` and prints `total, count : (6, 4)`. The total of the evens is right, $2 + 4$, but four numbers were counted, not two. That's the observation that rules out half the code: the bug is in how `count` grows. The second session confirms the hypothesis with a conditional breakpoint that stops only for odd numbers: at `n = 1`, `count` is about to become 1, although 1 isn't even. The line-by-line trace shows the same thing without a debugger, for `[1, 2]`: after `if n % 2 == 0:` is false for 1, the next line run is `count += 1`, because it isn't indented under the `if`. The fix is one level of indentation.

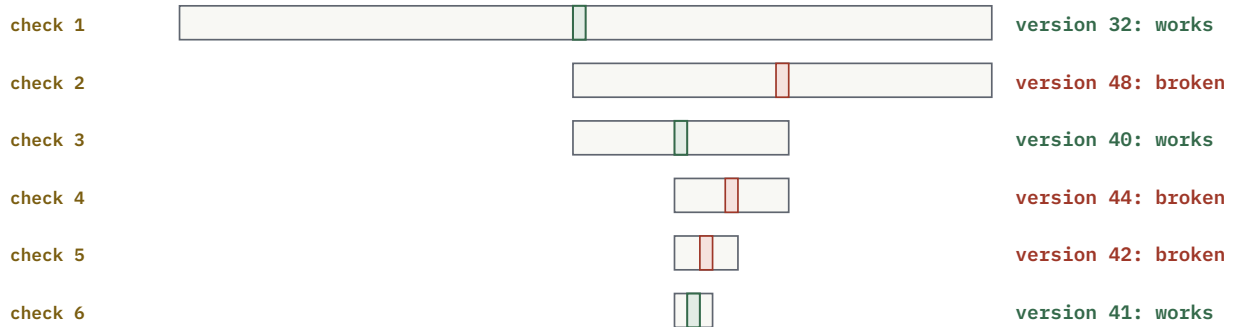
That kind of bug, where the code is plausible and the output is wrong, is exactly what reading code finds slowly and running it under a debugger finds quickly. `print` statements can do the same job, and for a quick check they're fine, but they need editing the code, rerunning, and remembering to remove them, and you can only print what you thought to print in advance. At a breakpoint you can ask any question that occurs to you. Get comfortable with one debugger early; it's one of the highest-return skills in this stage.

When a program crashes, you can debug it after the fact. `python -m pdb -c continue script.py` runs the script and, if it raises an uncaught exception, drops into the debugger at the point of the crash, with all the variables still there, which is called **post-mortem** debugging. In an interactive session, `import pdb; pdb.pm()` does the same for the last exception.

When you don't know where to look: bisect

Sometimes you know a thing worked last month and doesn't now, and nothing about the symptom points to a cause. Then search the history instead of the code. Find a version that works and one that doesn't, test the version halfway between, and keep the half that contains the change. Each test halves the range, so 64 versions take 6 tests and a thousand commits take 10. Git automates this as `git bisect`: mark one commit `good` and one `bad`, and Git checks out the midpoints for you to test, or runs a test script on each one by itself with `git bisect run`. The result is the exact commit that introduced the bug, usually small enough that the cause is obvious from the diff.

BISECTING 64 VERSIONS: THE BUG ARRIVED IN VERSION 42



result: 41 works and 42 is broken, so 42 introduced it; 6 checks for 64 versions

Bisection is binary search over history. Each check rules out half of the remaining versions, so the number of checks grows with the logarithm of the number of versions.

Bisection works on more than history. If a 10,000-line input file crashes the program, try the first half, then the half of whichever half fails, and in about 14 steps you have the one line that causes it. If a change touches twenty files and breaks something, revert half. The principle is the same as chapter 3's binary search: an invariant, "the cause is in this range", and a range that halves with every test.

Finally, explaining the problem out loud, line by line, to anyone or anything, often reveals the bug before you finish. Programmers call it **rubber duck debugging**, after a story of one who kept a rubber duck on the desk for the purpose. It works because explaining forces you to state the assumptions you were skipping over, and the bug is usually in one of them.

COMMON TRAPS

Changing code before you understand the bug, so the symptom moves and the cause stays.

Changing several things at once, so you can't tell which one mattered.

Debugging with a large input when a minimal reproduction would make the cause obvious.

Leaving `print` calls or `breakpoint()` in committed code. Search for them before committing.

Fixing a bug without adding a test for it, so it can return unnoticed.

EXERCISES

EXERCISE 12.1 The last three

`def recent_average(xs): return sum(xs[-3:-1]) / 3` should average the last three numbers. Before running it, write down the `pdb` commands you'd use to see what's being summed, then find and fix the bug.

SOLUTION

Put `breakpoint()` on a line before the return, or run under `pdb` with a breakpoint on the function, then `p xs[-3:-1]`. For `[1, 2, 3, 4, 5]` it shows `[3, 4]`: two numbers, not three, because a slice's end is excluded, and `-1` excludes the last item. `p sum(xs[-3:-1]) / 3` shows 2.33 instead of 4. The fix is `xs[-3:]`, which runs to the end. Dividing by `len(xs[-3:])` rather than 3 also makes the function correct for lists shorter than three, and raises `ZeroDivisionError` for an empty list, which should probably become a `ValueError` with a clear message. Printing the slice at a breakpoint took one command; guessing from the code alone means remembering the slice rules exactly, which is the step that went wrong in the first place.

EXERCISE 12.2 After the crash

A script crashes with `KeyError: 'email'` deep inside a loop over 5,000 records. Describe how to find which record caused it, without editing the script.

SOLUTION

Run it as `python -m pdb -c continue script.py`. It runs normally until the uncaught `KeyError`, then stops in post-mortem mode in the frame that raised it. `p record` shows the record being processed, or `p record.keys()` shows which keys it does have, which usually reveals the cause, such as `"Email"` with a capital, or a record with no email at all. `w` shows the call stack, and `u` moves up to the loop's frame, where `p i` or `p line_no` gives its position in the input. No print statements, no rerunning with guesses. The fix then belongs at the boundary, chapter 10: validate each record when it's read, and report the line number and the missing field.

EXERCISE 12.3 Automated bisection

A test, `tests/test_report.py::test_totals`, passes at the commit tagged `v1.4` and fails on `main`, with 300 commits in between. Write the commands to find the first bad commit automatically. Roughly how many test runs will it take?

SOLUTION

```
git bisect start main v1.4
git bisect run pytest -q tests/test_report.py::test_totals
git bisect reset
```

`git bisect start BAD GOOD` marks the two ends. `git bisect run` checks out a midpoint, runs the command, and treats exit status 0 as good and any other as bad (except 125, which means "can't test this commit, skip it"); pytest exits with 1 when a test fails, so it works as is. Each run halves the range, so 300 commits take about 9 runs, since $2^9 = 512$. Git then prints the first bad commit, and `reset` returns you to where you started. Bisection only works if the test is deterministic, which is one more reason chapter 11's advice about seeds and clocks matters, and if each commit builds and runs, which is a reason to keep commits small and working.

CHECKLIST

- ☐ Can you follow reproduce, observe, hypothesise, test, fix, and say why the order matters?
- ☐ Can you use `breakpoint()` and pdb's `p`, `n`, `s`, `c`, `b`, `w`, `u` and `d`?
- ☐ Can you set a conditional breakpoint inside a loop?
- ☐ Can you debug a crash after the fact with post-mortem mode?
- ☐ Can you bisect a history, an input file or a change to find the cause in logarithmic steps?

GO DEEPER

DOCS [pdb: The Python Debugger Python documentation](#)

DOCS [git-bisect: find the commit that introduced a bug Git documentation](#)

BOOK [Debugging: The 9 Indispensable Rules David Agans](#)

PART IV

Structure

Four chapters on organising programs that are too big to read in one sitting. Classes bundle data with the code that keeps it valid. Composition and interfaces decide how classes fit together, and when inheritance helps or quietly couples everything to everything. References and aliasing explain what's actually in memory when two parts of a program share an object, the source of the bugs that experienced programmers still write. And modules, packages and environments are how code is split across files, shared between projects, and installed without one project breaking another.

Objects and classes

3 exercises

A class defines a new type: what data its objects hold, and what operations they support. Its real value is that the operations can keep the data valid. A booking can refuse to end before it starts, and a calendar can refuse to double-book a room, so no other code has to remember those rules. This chapter builds a small booking system, shows what a method call really is, demonstrates the class-attribute trap that shares one list between every instance, and uses dataclasses for the many classes that are mostly data.

Classes, instances and methods

A **class** is a blueprint; an **instance** is an object made from it. `class Booking:` defines the type, and `Booking("Oak", 540, 600)` calls the class to make an instance, which runs the special method `__init__` to set it up. Inside a method, `self` is the instance the method was called on, and `self.room = room` stores an **attribute** on it. A **method** is a function defined in the class, and calling it on an instance, `b.overlaps(other)`, is exactly the same as calling the class's function with the instance as the first argument, `Booking.overlaps(b, other)`. The example prints both. There's no magic in `self`; it's just the name, by convention, of that first parameter.

The example's `Booking` has an **invariant**, like chapter 3's loops: every booking's start is before its end, within one day. `__init__` checks it and raises `ValueError` if it doesn't hold, so an invalid booking can't exist. `minutes` is a **property**, a method that's read like an attribute, `b.minutes`, without parentheses. It's computed from `start` and `end` each time, so it can never disagree with them, which a stored `minutes` attribute could after someone changed `end`. And `__repr__` controls how the object prints, which matters more than it seems: `Booking('Oak', 09:00-10:00)` in a debugger or a log tells you something, and the default `<Booking object at 0x104f...>` doesn't.

EXAMPLE

classes.py · a `Booking` that guards its own invariant, a `Calendar` that refuses clashes, what a method call is, the shared class-attribute trap, and a frozen ordered dataclass

```
from dataclasses import dataclass, FrozenInstanceError

# 1. A class bundles data with the operations that keep it valid
class Booking:
    """A room booked from start to end, in minutes since midnight."""
    def __init__(self, room, start, end):
        if not 0 <= start < end <= 24 * 60:                # the invariant every Booking
            keeps
            raise ValueError(f"bad times {start}-{end}")
        self.room, self.start, self.end = room, start, end
    @property
    def minutes(self):                                     # computed, so it can't go stale
```

```

        return self.end - self.start
    def overlaps(self, other):
        return self.room == other.room and self.start < other.end and other.start < self.end
    def __repr__(self):
        fmt = lambda m: f"{m // 60:02d}:{m % 60:02d}"
        return f"Booking({self.room!r}, {fmt(self.start)}-{fmt(self.end)})"

class Calendar:
    def __init__(self): self._bookings = [] # the underscore: "don't touch this"
    def book(self, room, start, end):
        new = Booking(room, start, end)
        clash = next((b for b in self._bookings if b.overlaps(new)), None)
        if clash: raise ValueError(f"{new} clashes with {clash}")
        self._bookings.append(new)
        return new
    def busiest(self):
        totals = {}
        for b in self._bookings: totals[b.room] = totals.get(b.room, 0) + b.minutes
        return max(totals.items(), key=lambda kv: kv[1])

cal = Calendar()
requests = [("Oak", 9 * 60, 10 * 60), ("Oak", 10 * 60, 11 * 60 + 30), ("Elm", 9 * 60 + 30, 11 * 60),
            ("Oak", 11 * 60, 12 * 60), ("Elm", 13 * 60, 12 * 60)]
for room, start, end in requests:
    try: print(f"booked {cal.book(room, start, end)}")
    except ValueError as e: print(f"refused: {e}")
print(f"busiest room: {cal.busiest()}")

# 2. A method call is a function call with the object passed as self
b = Booking("Ash", 600, 660)
print(f"\nb.minutes = {b.minutes}; b.overlaps(b) = {b.overlaps(b)}, "
      f"and Booking.overlaps(b, b) = {Booking.overlaps(b, b)}: the same call")

# 3. Attributes set on the class are shared by every instance
class Team:
    members = [] # one list, on the class
    def __init__(self, name): self.name = name
    def add(self, person): self.members.append(person)
red, blue = Team("red"), Team("blue")
red.add("Ana")
print(f"\nafter red.add('Ana'): blue.members = {blue.members} (shared!)")
class FixedTeam:
    def __init__(self, name): self.name, self.members = name, [] # one list per instance
red, blue = FixedTeam("red"), FixedTeam("blue")
red.members.append("Ana")
print(f"with members set in __init__: blue.members = {blue.members}")

# 4. For classes that are mostly data, a dataclass writes __init__, __repr__ and __eq__ for you
@dataclass(frozen=True, order=True)
class Version:
    major: int
    minor: int = 0
    patch: int = 0
releases = [Version(1, 10), Version(1, 2, 3), Version(2), Version(1, 2)]
print(f"\n{sorted(releases)}")
print(f"Version(1, 2) == Version(1, 2, 0): {Version(1, 2) == Version(1, 2, 0)}; "
      f"compared as strings, '1.10' < '1.2': {'1.10' < '1.2'}")
try: releases[0].major = 9
except FrozenInstanceError as e: print(f"frozen: {e}")

```

OUTPUT

```

booked Booking('Oak', 09:00-10:00)
booked Booking('Oak', 10:00-11:30)

```

```

booked Booking('Elm', 09:30-11:00)
refused: Booking('Oak', 11:00-12:00) clashes with Booking('Oak', 10:00-11:30)
refused: bad times 780-720
busiest room: ('Oak', 150)

b.minutes = 60; b.overlaps(b) = True, and Booking.overlaps(b, b) = True: the same call

after red.add('Ana'): blue.members = ['Ana'] (shared!)
with members set in __init__: blue.members = []

[Version(major=1, minor=2, patch=0), Version(major=1, minor=2, patch=3), Version(major=1,
    minor=10, patch=0), Version(major=2, minor=0, patch=0)]
Version(1, 2) == Version(1, 2, 0): True; compared as strings, '1.10' < '1.2': True
frozen: cannot assign to field 'major'

```

Responsibilities

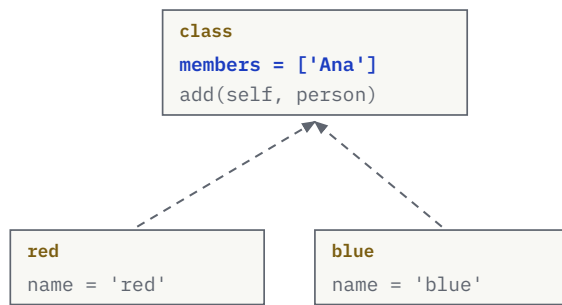
Notice how the work is divided. A `Booking` knows about itself: whether its times are valid and whether it overlaps another booking. A `Calendar` knows about the collection: it holds the bookings, refuses a new one that clashes, and answers questions about all of them, such as which room is busiest. Neither reaches into the other's data. That's the idea of **encapsulation**: each class owns its data and the rules for it, and other code goes through its methods. Python doesn't enforce privacy, but a leading underscore, `self._bookings`, is the universal signal that an attribute is internal and may change, so outside code should leave it alone.

When should you write a class instead of functions and dicts? When data and the rules for that data travel together, and you'd otherwise have to repeat the rules everywhere the data is used. A dict `{"room": "Oak", "start": 600, "end": 540}` is happy to hold an impossible booking; a `Booking` isn't. When you have no rules to enforce, only a few fields, a dataclass or even a plain tuple is enough, and a module of functions is often better than a class with no data at all.

Class attributes are shared

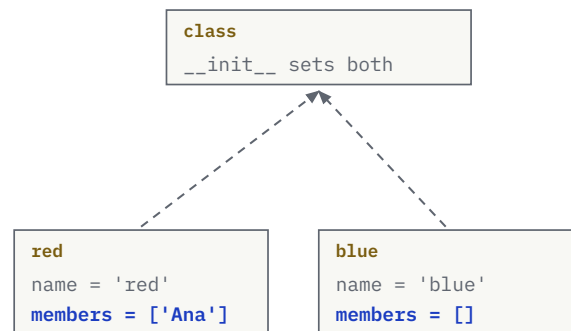
An attribute assigned in the class body, outside any method, belongs to the class, not to any instance. When you read `blue.members`, Python looks on the instance first and, finding nothing, looks on the class. So in the example's `Team`, where `members = []` is written in the class body, every team shares one list, and adding a member to red adds it to blue. It's the same trap as chapter 4's mutable default argument, for the same reason: the list was created once, when the class was defined.

TEAM: MEMBERS ON THE CLASS



`blue.members` isn't on `blue`, so Python looks on the class

FIXEDTEAM: MEMBERS PER INSTANCE



each instance finds its own list first

Where Python finds `blue.members`. An attribute missing from the instance is looked up on the class, so a mutable class attribute is quietly shared by every instance.

The fix is to create mutable attributes in `__init__`, so each instance gets its own. Class attributes are fine for constants that every instance should share and nobody changes, like a default rate or a table of valid values.

Dataclasses

Many classes are mostly data, and writing `__init__`, `__repr__` and `__eq__` for each is tedious and error-prone. The `@dataclass` decorator generates them from a list of annotated fields. With `order=True` it also generates comparisons, which compare the fields in order, so the example's `Version` objects sort correctly: 1.2.0, 1.2.3, 1.10.0, 2.0.0. Compared as strings, "1.10" would sort before "1.2", because string comparison goes character by character and "1" < "2". With `frozen=True`, assigning to a field raises an error, so a frozen dataclass is a value like a number or a string: safe to share, and usable as a dict key, because the generated `__hash__` agrees with `__eq__`, as chapter 7 requires. Prefer frozen dataclasses unless the object genuinely needs to change.

COMMON TRAPS

Mutable class attributes, such as `members = []` in the class body, shared by every instance.

Storing a derived value that can drift out of date. Compute it with a property.

Constructors that accept invalid data and leave validation to every user of the class.

Reaching into another object's underscore attributes instead of asking it through a method.

Writing a class for something that has no data or no rules, where a function would do.

EXERCISE 13.1 Exposing bookings safely

Add `Calendar.bookings_for(room)`, returning that room's bookings sorted by start time. What goes wrong if it returns `self._bookings` itself, and how do you prevent it?

SOLUTION

```
def bookings_for(self, room):
    return sorted((b for b in self._bookings if b.room == room), key=lambda b:
                  b.start)
```

Returning the internal list itself would let a caller do `cal.bookings_for("Oak").append(clashing_booking)`, or `.clear()`, and break the calendar's rule that bookings never clash, without going through `book`. Returning a new list, which `sorted` always does, means callers can do whatever they like with the result without affecting the calendar. The `Booking` objects inside are shared, so a caller could still change `b.end` on one; making `Booking` a frozen dataclass closes that last gap, and then the only way to change the calendar is through its methods, which check the rules.

EXERCISE 13.2 Parsing versions

Write `Version.parse(text)`, a class method that turns `"1.10.2"` into `Version(1, 10, 2)` and raises `ValueError` for anything else. Show that sorting parsed versions gives the right order where sorting the strings doesn't.

SOLUTION

```
@dataclass(frozen=True, order=True)
class Version:
    major: int
    minor: int = 0
    patch: int = 0
    @classmethod
    def parse(cls, text):
        parts = text.split(".")
        if not 1 <= len(parts) <= 3 or not all(p.isdigit() for p in parts):
            raise ValueError(f"not a version: {text!r}")
        return cls(*map(int, parts))

tags = ["1.10.0", "1.2.3", "2.0", "1.2"]
assert sorted(tags) == ["1.10.0", "1.2", "1.2.3", "2.0"] # wrong
assert [str(v.major) + "." + str(v.minor) for v in sorted(map(Version.parse, tags))]
       == \
       ["1.2", "1.2", "1.10", "2.0"] # right
```

A `@classmethod` receives the class, `cls`, instead of an instance, which makes it the standard way to write alternative constructors such as `parse`, `from_json` or `datetime.fromisoformat`. The validation follows chapter 10: reject anything that isn't one to three numeric parts, with the bad text in the message. Sorting strings puts 1.10 before 1.2; sorting the parsed values compares numbers field by field and gets it right. Real version strings have more forms, such as `1.0.0rc1`, which is why Python projects use the `packaging` library's `Version` for this.

EXERCISE 13.3 A bank account's invariant

Design an `Account` class whose balance can never go below zero. Which methods does it need, what do they check, and why should `balance` be a read-only property?

SOLUTION

```
class Account:
    def __init__(self, owner, balance=0):
        if balance < 0: raise ValueError("opening balance can't be negative")
        self.owner, self._balance = owner, balance
    @property
    def balance(self): return self._balance
    def deposit(self, amount):
        if amount <= 0: raise ValueError("deposit must be positive")
        self._balance += amount
    def withdraw(self, amount):
        if amount <= 0: raise ValueError("withdrawal must be positive")
        if amount > self._balance: raise ValueError(f"insufficient funds:
            {self._balance} available")
        self._balance -= amount
```

The invariant, $\text{balance} \geq 0$, is established by `__init__` and maintained by every method that changes the balance, each of which checks before changing anything, so a failed withdrawal leaves the account exactly as it was. `balance` is a property with no setter, so `acct.balance = -50` raises `AttributeError`: the only ways to change the balance are the methods that enforce the rule. Amounts should be integers counting pence or cents, chapter 1, not floats. A real system would also record each transaction, so that the balance can be audited as the sum of the history rather than trusted as a number.

CHECKLIST

- ☐ Can you define a class with `__init__`, methods, a property and a useful `__repr__`?

- ☐ Can you explain what `self` is and rewrite a method call as a plain function call?
- ☐ Can you enforce an invariant in a constructor and keep it in every method?
- ☐ Can you explain the difference between class and instance attributes, and the shared-list trap?
- ☐ Can you use frozen, ordered dataclasses and say when a class isn't needed at all?

GO DEEPER

DOCS [The Python Tutorial: Classes Python documentation](#)

DOCS [dataclasses: Data Classes Python documentation](#)

BOOK [Composing Programs, 2.5: Object-Oriented Programming John DeNero](#)

Composition and interfaces

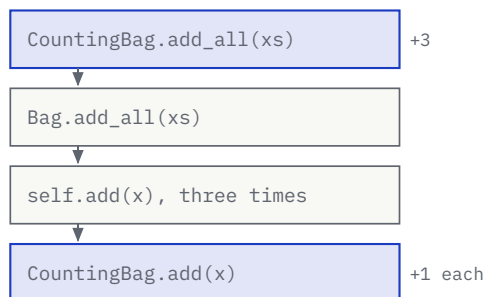
3 exercises

Inheritance looks like the natural way to reuse code: a counting bag is a bag that also counts, so subclass it. But a subclass depends on how its parent works inside, not just on what it does, and that dependency breaks in ways that are hard to see. Composition, an object that holds another and delegates to it, depends only on the public behaviour. This chapter shows a subclass that counts every item twice, a square that breaks code written for rectangles, and the alternatives: composition, interfaces defined by what callers need, and functions passed in where one behaviour varies.

Inheritance couples you to the inside

Inheritance makes a new class from an existing one: `class CountingBag(Bag)` gets all of `Bag`'s methods, and can **override** some of them, calling the parent's version with `super()`. It's appealing because the subclass is short. The example's `CountingBag` overrides both `add` and `add_all` to count, and adding three items reports six. The reason is inside `Bag`: its `add_all` happens to call `self.add` for each item, and since `self` is the `CountingBag`, that's the counting `add`, so every item is counted once by `add_all` and again by `add`. Nothing in `Bag`'s public behaviour said so; it's an implementation detail. If the author of `Bag` later changed `add_all` to append directly, the count would become correct, and a subclass that had been "fixed" to compensate would break.

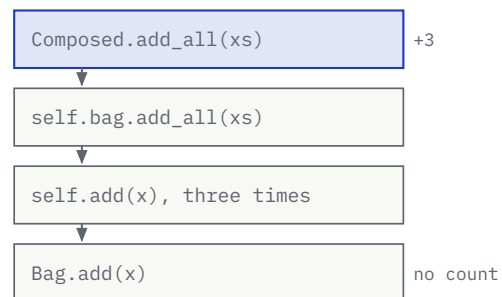
INHERITED



counter: 6 for 3 items

inside Bag, `self` is still the `CountingBag`,
so its overridden `add` runs too

COMPOSED



counter: 3 for 3 items

inside the Bag, `self` is the Bag: no overrides

Why the inherited counter double-counts. Inside the parent's code, `self` is still the subclass, so the parent's internal calls reach the subclass's overrides. Composition has no such back door.

This is called the **fragile base class problem**, and it's why the 1994 book *Design Patterns* advised "favour object composition over class inheritance". The composed version, `CountingBagComposed`, holds a `Bag` and forwards each call to it. Inside the bag, `self` is the bag, so its internal calls never reach the counter, and the count is right whatever `Bag` does inside. The cost is that every method you want to expose must be written out, one line each; the benefit is that you depend only on `Bag`'s public methods, which its author has promised to keep working.

EXAMPLE

composition.py · a subclass that double-counts and the composed fix, a Square that breaks Rectangle code, a Protocol for an interface, an abstract base class, and behaviour passed as a function

```
from abc import ABC, abstractmethod
from typing import Protocol, runtime_checkable

# 1. Inheritance ties a subclass to how its parent works inside, not just to what it does
class Bag:
    def __init__(self): self.items = []
    def add(self, x): self.items.append(x)
    def add_all(self, xs):
        for x in xs: self.add(x) # an implementation detail: it calls add
class CountingBag(Bag):
    """Counts how many items were ever added, by overriding both methods."""
    def __init__(self): super().__init__(); self.added = 0
    def add(self, x): self.added += 1; super().add(x)
    def add_all(self, xs): self.added += len(xs); super().add_all(xs)
bag = CountingBag()
bag.add_all(["a", "b", "c"])
print(f"inherited: added 3 items, counter says {bag.added}") # add_all counted, then add
# counted again

class CountingBagComposed:
    """The same feature by holding a Bag instead of being one."""
    def __init__(self): self.bag, self.added = Bag(), 0
    def add(self, x): self.added += 1; self.bag.add(x)
    def add_all(self, xs): self.added += len(xs); self.bag.add_all(xs)
    @property
    def items(self): return self.bag.items
bag = CountingBagComposed()
bag.add_all(["a", "b", "c"])
print(f"composed: added 3 items, counter says {bag.added}")

# 2. A subclass must work wherever its parent does. A Square is not a substitutable Rectangle.
class Rectangle:
    def __init__(self, w, h): self.w, self.h = w, h
    def set_width(self, w): self.w = w
    def area(self): return self.w * self.h
class Square(Rectangle):
    def __init__(self, side): super().__init__(side, side)
    def set_width(self, w): self.w = self.h = w # keeps it square, breaks the parent's
# promise
def stretch(rect):
    rect.set_width(rect.w * 2) # code written for Rectangle expects this
    return rect.area()
print(f"\nstretch(Rectangle(2, 3)) = {stretch(Rectangle(2, 3))} (expected 12); "
      f"stretch(Square(3)) = {stretch(Square(3))} (expected 18)")

# 3. An interface is what callers rely on. In Python it can be just a shape, checked by a
# Protocol.
```

```

@runtime_checkable
class HasArea(Protocol):
    def area(self) -> float: ...
class Circle:
    def __init__(self, r): self.r = r
    def area(self): return round(3.14159 * self.r ** 2, 2)
shapes = [Rectangle(2, 3), Circle(1), Square(2)]
print(f"\ntotal area: {sum(s.area() for s in shapes)}; "
      f"all satisfy HasArea: {all(isinstance(s, HasArea) for s in shapes)}; "
      f"a str does: {isinstance('x', HasArea)}")

# 4. When you do inherit, an abstract base class makes the contract explicit
class Exporter(ABC):
    @abstractmethod
    def write(self, rows): ...
    def export(self, rows): return self.write(sorted(rows)) # shared logic, once
class CsvExporter(Exporter):
    def write(self, rows): return "\n".join(",".join(map(str, r)) for r in rows)
class BrokenExporter(Exporter): pass
print(f"\n{CsvExporter().export([(2, 'b'), (1, 'a')])!r}")
try: BrokenExporter()
except TypeError as e: print(f"BrokenExporter() -> TypeError: {e}")

# 5. Often the thing that varies is one behaviour: pass a function instead of making subclasses
def price(items, rule=lambda total: total):
    return rule(sum(items))
rules = {"none": lambda t: t, "10% off": lambda t: round(t * 0.9, 2),
        "5 off over 50": lambda t: t - 5 if t > 50 else t}
print(f"\n[{(name, price([20, 35], rule)) for name, rule in rules.items()}]")

```

OUTPUT

```

inherited: added 3 items, counter says 6
composed:  added 3 items, counter says 3

stretch(Rectangle(2, 3)) = 12 (expected 12); stretch(Square(3)) = 36 (expected 18)

total area: 13.14; all satisfy HasArea: True; a str does: False

'1,a\n2,b'
BrokenExporter() -> TypeError: Can't instantiate abstract class BrokenExporter without an
    implementation for abstract method 'write'

[('none', 55), ('10% off', 49.5), ('5 off over 50', 50)]

```

A subclass must be substitutable

A subclass is a promise: anywhere the program uses a `Rectangle`, a `Square` will do. That promise is the **Liskov substitution principle**, after Barbara Liskov, who stated it in 1987. The example's `Square` breaks it. Mathematically a square is a rectangle, so inheritance seems right, but `Rectangle`'s `set_width` promises to change only the width, and a square can't keep that promise and stay square. Code written for rectangles, the example's `stretch`, doubles the width and expects double the area; with a square it gets four times the area. The lesson is that inheritance should model "behaves like", not "is a kind of" in everyday language. Here the trouble comes from mutation: if rectangles were immutable, with no `set_width`, a square would be a perfectly good one, which exercise 14.2 explores.

Interfaces: what callers need

An **interface** is the set of operations a caller relies on. Python doesn't require you to declare one. A function that calls `shape.area()` works with any object that has an `area` method, whatever its class, which is called **duck typing**: if it has the methods you need, it'll do. The example's `Circle` isn't related to `Rectangle` at all, and `total_area` happily mixes them. To make the interface explicit, for readers and for type checkers, write a `Protocol`: a class listing the methods required, which any class with those methods satisfies automatically, without inheriting from it. With `@runtime_checkable`, `isinstance(x, HasArea)` even checks it at run time, though only that the method exists, not its signature.

Sometimes you do want a family of classes with shared logic, and then an **abstract base class** states the contract. The example's `Exporter` marks `write` as abstract and implements `export` once on top of it; subclasses supply only `write`, and one that forgets can't even be instantiated, so the mistake appears at once instead of when `export` is first called. That's inheritance used well: the base class is designed for extension, with a documented hook, rather than a class someone happened to subclass. The standard library is full of these, and every exception class you define, chapter 10, is the same pattern.

Often, pass a function

When the only thing that varies between "kinds" of an object is one piece of behaviour, a class hierarchy is heavy machinery. The example's pricing has three rules, and rather than `NoDiscountPrice`, `PercentOffPrice` and `ThresholdPrice` subclasses, the rule is a function passed to `price`. Adding a fourth rule is one line, rules can be stored in a dict, chosen from configuration and tested alone. It's chapter 4's "functions are values" applied to design. Reach for a class when the behaviour needs its own data and several methods; reach for a function when it's one operation.

COMMON TRAPS

Subclassing a class you don't control and depending on which of its methods call which.

Modelling "is a kind of" with inheritance when the subclass can't keep the parent's promises.

Deep hierarchies, where understanding one method means reading five classes.

Checking `type(x) == Something` instead of relying on the methods you need.

A class hierarchy for one varying behaviour, where passing a function would do.

EXERCISES

EXERCISE 14.1 Where the log goes

A `Report` class needs to write its output to the screen, to a file, or into a list for tests. Compare a design with three subclasses, `ScreenReport`, `FileReport` and `TestReport`, with a design where `Report` takes a writer. Which is easier to extend and test?

SOLUTION

```
class Report:
    def __init__(self, rows, write=print):
        self.rows, self.write = rows, write
    def render(self):
        for r in self.rows:
            self.write(f"{r['name']:10} {r['total']:>8.2f}")

lines = []
Report([{"name": "Ana", "total": 28.5}], write=lines.append).render()
assert lines == ["Ana", "28.50"]
```

With subclasses, each destination duplicates or inherits the report logic, and a fourth destination, or a report that writes to two places, means another subclass. Worse, the report's formatting and its destination are tangled together, so a change to one risks the other. Passing `write` separates them: the report formats, and the caller decides where lines go. `print` is the default, `open(path, "w").write` wrapped to add newlines works for files, and a list's `append` captures output for tests with no mocking at all. This is chapter 11's dependency injection again. Composition of small parts beats a hierarchy whenever the parts vary independently.

EXERCISE 14.2 Squares that substitute

Redesign `Rectangle` and `Square` so that a `Square` can be used anywhere a `Rectangle` is, and `stretch` still makes sense.

SOLUTION

```
@dataclass(frozen=True)
class Rectangle:
    w: float
    h: float
    def area(self): return self.w * self.h
    def with_width(self, w): return Rectangle(w, self.h)

def Square(side): return Rectangle(side, side)

def stretch(rect): return rect.with_width(rect.w * 2).area()
assert stretch(Square(3)) == 18
```

The problem was a method that mutated the object in a way a square couldn't honour. Make rectangles immutable, and "changing the width" returns a new rectangle; a square stretched is simply a rectangle that isn't square any more, which is exactly right. Here a square isn't a separate class at all, just a convenient way to build a rectangle with equal sides; if squares needed special behaviour, a `Square` class could add it without overriding anything `Rectangle` promises. Immutability removes a whole family of substitution problems, which is one reason chapter 13 recommends frozen dataclasses by default.

EXERCISE 14.3 Anything you can read

Write `count_lines(source)` that works with any object having a `readline` method, define a `Protocol` for it, and test it with `io.StringIO` instead of a real file.

SOLUTION

```
import io
from typing import Protocol

class LineSource(Protocol):
    def readline(self) -> str: ...

def count_lines(source: LineSource) -> int:
    n = 0
    while source.readline():
        n += 1
    return n

assert count_lines(io.StringIO("a\nb\nc\n")) == 3
```

`count_lines` calls only `readline`, so it works with an open file, a network stream, `sys.stdin` or an `io.StringIO`, none of which share a useful base class. The `Protocol` writes down that requirement, so a type checker can confirm each caller passes something suitable, and a reader knows exactly what the function needs. Testing with `StringIO` needs no files on disk and runs instantly. Accepting the narrowest interface that does the job is chapter 8 of *The Engineer's Press's* advice from the type-system side: ask for `Iterable` rather than `list`, for `LineSource` rather than a file.

CHECKLIST

- ☐ Can you explain the fragile base class problem with an example?

- ☐ Can you replace a subclass with composition and delegation?
- ☐ Can you state the Liskov substitution principle and spot a violation?
- ☐ Can you define an interface with a `Protocol`, and a contract with an abstract base class?
- ☐ Can you choose between a class hierarchy and passing a function?

GO DEEPER

DOCS [Protocols and structural subtyping Python typing specification](#)

DOCS [abc: Abstract Base Classes Python documentation](#)

ARTICLE [The Composition Over Inheritance Principle Brandon Rhodes](#)

References, mutation and aliasing

3 exercises

*Every Python object lives in one place in memory, and every name, list slot and attribute that refers to it holds a reference, a pointer to that place. When two references point to the same mutable object, a change through one is visible through the other. That's **aliasing**, and it's sometimes exactly what you want and sometimes the cause of a bug that seems impossible. This chapter draws what's actually in memory for the classic aliasing bugs, explains copying, identity and how Python decides an object is no longer needed, and connects it to the stack and the heap that CS50 shows you in C.*

The stack, the heap and references

Chapter 5 described the call stack: each function call gets a frame holding its local variables. In Python, what a frame holds isn't the objects themselves but **references** to them. The objects live in the **heap**, a large region of memory where they're created and stay for as long as anything refers to them, independent of which function made them. That's why a function can create a list and return it: the frame disappears, but the list doesn't, because the caller now holds a reference. In C you handle these addresses directly, as pointers, which is why CS50's week on memory is worth doing even if you'll write Python: Python's references are pointers that the language manages for you.

Passing an argument copies the reference, not the object, so the function's parameter and the caller's variable refer to the same object. This is sometimes called **call by sharing**. The example's `add_bonus` mutates the list it's given, and the caller sees the change; `reset` rebinds its own parameter to a new list, which the caller never sees. That's chapter 2's distinction between mutating and rebinding, crossing a function call. A function that mutates its arguments should say so in its name or docstring, and most functions shouldn't do it at all.

EXAMPLE

memory.py · call by sharing, the grid and fromkeys aliasing bugs, shallow and deep copies, identity, objects freed and cycles collected, and a tuple that changes

```
import copy, gc, weakref

def picture(**names):
    """Which names share an object: label each distinct object #1, #2, ... (ids vary between runs)."""
    labels = {}
    for name, obj in names.items(): labels.setdefault(id(obj), f"#{len(labels) + 1}")
    return " " + " ".join(f"{name} -> {labels[id(obj)]} {obj!r}" for name, obj in
        names.items())
```

1. Calling a function passes a reference to the object, not a copy of it

```

def add_bonus(scores): scores.append(100)           # mutates the caller's list
def reset(scores): scores = []                     # rebinds the local name only
mine = [70, 80]
add_bonus(mine); reset(mine)
print(f"after add_bonus and reset: mine = {mine}")

# 2. The grid bug: * copies references, so every row is the same list
grid = [[0] * 3] * 3
grid[0][0] = 1
print(f"\n[[0] * 3] * 3, then grid[0][0] = 1: {grid}")
print(picture(row0=grid[0], row1=grid[1], row2=grid[2]))
grid = [[0] * 3 for _ in range(3)]                 # a new list for each row
grid[0][0] = 1
print(f"with a comprehension: {grid}")
groups = dict.fromkeys(["a", "b"], [])              # the same trap: one list for every key
groups["a"].append("x")
print(f"dict.fromkeys(keys, []) then groups['a'].append('x'): {groups}")

# 3. Copies: a shallow copy shares the insides; a deep copy copies all the way down
team = {"name": "red", "members": ["Ana", "Ben"]}
shallow, deep = copy.copy(team), copy.deepcopy(team)
team["members"].append("Cy")
print(f"\nafter appending to the original's members: shallow {shallow['members']}, deep {deep['members']}")
print(picture(original=team["members"], shallow=shallow["members"], deep=deep["members"]))

# 4. == compares values; is compares identity
a, b = [1, 2], [1, 2]
print(f"\na == b: {a == b}; a is b: {a is b}; a is a: {a is a}; use 'is' only for None and sentinels")

# 5. An object lives while something refers to it, and is freed when nothing does
class Node:
    def __init__(self, name): self.name, self.other = name, None
n = Node("solo"); probe = weakref.ref(n)            # a weak reference doesn't keep it alive
del n
print(f"\nafter del, the lone node is gone: {probe() is None}")
x, y = Node("x"), Node("y")
x.other, y.other = y, x                             # a cycle: each keeps the other's count above zero
probe = weakref.ref(x)
gc.disable(); del x, y
print(f"a cycle after del: still alive? {probe() is not None}")
gc.collect(); gc.enable()
print(f"after the cycle collector runs: still alive? {probe() is not None}")

# 6. A puzzle: a tuple can't change, but the list inside it can
t = ([1, 2],)
try: t[0] += [3]                                     # extends the list, then fails to reassign
    t[0]
except TypeError as e: print(f"\nt[0] += [3] raised TypeError: {e}")
print(f"and yet t = {t}")

```

OUTPUT

```

after add_bonus and reset: mine = [70, 80, 100]

[[0] * 3] * 3, then grid[0][0] = 1: [[1, 0, 0], [1, 0, 0], [1, 0, 0]]
  row0 -> #1 [1, 0, 0]   row1 -> #1 [1, 0, 0]   row2 -> #1 [1, 0, 0]
with a comprehension: [[1, 0, 0], [0, 0, 0], [0, 0, 0]]
dict.fromkeys(keys, []) then groups['a'].append('x'): {'a': ['x'], 'b': ['x']}

after appending to the original's members: shallow ['Ana', 'Ben', 'Cy'], deep ['Ana', 'Ben']
  original -> #1 ['Ana', 'Ben', 'Cy']   shallow -> #1 ['Ana', 'Ben', 'Cy']   deep -> #2
    ['Ana', 'Ben']

```

```

a == b: True; a is b: False; a is a: True; use 'is' only for None and sentinels

after del, the lone node is gone: True
a cycle after del: still alive? True
after the cycle collector runs: still alive? False

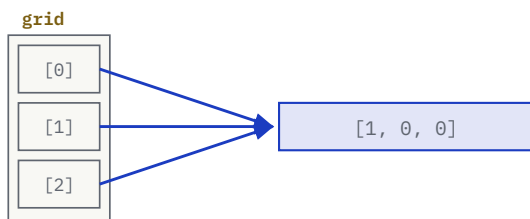
t[0] += [3] raised TypeError: 'tuple' object does not support item assignment
and yet t = ([1, 2, 3],)

```

Aliasing bugs

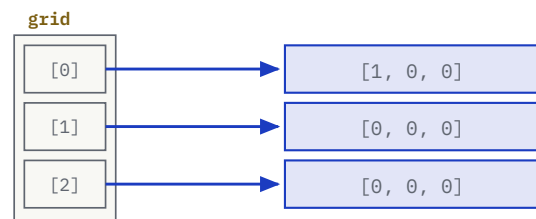
The best-known aliasing bug is building a grid with `[[0] * 3] * 3`. The inner expression makes one list of three zeros. Multiplying the outer list by 3 repeats its contents, which is a reference to that one list, so the grid holds three references to the same row. Setting `grid[0][0] = 1` changes the only row there is, and all three "rows" show it. The example's `picture` helper labels each distinct object, and all three rows are object #1. A comprehension, `[[0] * 3 for _ in range(3)]`, evaluates `[0] * 3` three times and makes three separate rows.

`[[0] * 3] * 3`: ONE ROW, THREE REFERENCES



`grid[0][0] = 1` changes the only row; every index sees it

A COMPREHENSION: THREE ROWS



`grid[0][0] = 1` changes row 0 only

What `[[0] * 3] * 3` builds. Multiplying a list copies references, not the objects they point to, so all three slots share one row.

The same mistake hides in other places. `dict.fromkeys(keys, [])` gives every key the same list. A mutable default argument, chapter 4, is one list shared by every call, and a mutable class attribute, chapter 13, is one list shared by every instance. Storing a list you were given in an object, `self.items = items`, means the caller can still change it behind the object's back. In every case the fix is the same: create a fresh object each time it's needed, or take a copy at the boundary.

Copies

There are two kinds of copy. A **shallow copy**, `copy.copy(x)`, `list(xs)`, `xs[:]` or `d.copy()`, makes a new outer container holding the same references as the original. A **deep copy**, `copy.deepcopy(x)`, copies the container and, recursively, everything inside it. For a flat list of numbers or strings the difference doesn't matter, since the items are immutable. For nested data it does: the example's shallow copy of a team still shares the members list with the original, so appending to the original's members shows up in the copy, while the deep copy is fully independent.

Deep copies have costs: they take time and memory proportional to everything reachable, and they can copy things that should stay shared, such as an open file or a reference to a shared configuration. Often a better approach than copying defensively is not mutating in the first place: build new values from old ones, as chapter 4's pure functions do and chapter 19 of *The Engineer's Press* explores, and sharing becomes harmless.

Identity, and when objects are freed

`a == b` asks whether two objects have equal values; `a is b` asks whether they're the same object. Two lists built separately with the same contents are equal but not identical. Use `is` only to compare with `None` and with sentinel objects you created for the purpose. `is` on numbers or strings gives results that depend on interpreter details, since CPython reuses some small integers and strings and not others, so `x is 5` is a bug even when it happens to work.

Python frees an object when nothing refers to it any more. CPython keeps a **reference count** on every object and frees it the moment the count reaches zero, which is why the example's lone node is gone as soon as its only name is deleted; a `weakref` lets you watch an object without keeping it alive. Objects that refer to each other in a **cycle** keep each other's counts above zero, so CPython also runs a cycle collector from time to time, and the example's cycle survives the deletion and is freed only when `gc.collect()` runs. Chapter 13 of *The Engineer's Press* explains these collectors in depth. For everyday code, the consequence is that Python programs leak memory not by forgetting to free but by remembering too much: a cache, a global list or a long-lived object that keeps references to things nobody needs.

The example ends with a puzzle that ties it together. A tuple can't change, so `t[0] += [3]` must fail, and it does, with a `TypeError`. But afterwards the list inside the tuple has grown anyway. `+=` on a list first extends the list in place, which succeeds because lists are mutable, and then assigns the result back into `t[0]`, which fails because tuples aren't. The tuple's immutability means its references can't change; the objects they point to can.

COMMON TRAPS

Building nested lists with `*`, so every row is the same list.

Storing a caller's list or dict in an object without copying it.

Using a shallow copy of nested data and expecting it to be independent.

Using `is` to compare numbers or strings. Use `==`.

Caches and registries that only grow, holding references that keep memory alive forever.

EXERCISE 15.1 Predict the aliasing

What does this print? Draw the objects before answering.

```
a = [1, 2]
b = [a, a]
c = b[:]
c[0].append(3)
c[1] = [9]
print(a, b, c)
```

SOLUTION

It prints `[1, 2, 3] [[1, 2, 3], [1, 2, 3]] [[1, 2, 3], [9]]`. `b` holds two references to the same list `a`. `c = b[:]` is a shallow copy: a new outer list, holding the same two references. `c[0].append(3)` follows `c`'s first reference to `a` and mutates it, so `a`, both of `b`'s slots, and `c`'s first slot all show `[1, 2, 3]`. `c[1] = [9]` replaces the reference in `c`'s own second slot, which affects `c` only, since its outer list is its own. The pattern is general: mutation through a shared reference is seen everywhere; replacing a reference is seen only in the container you replaced it in.

EXERCISE 15.2 Updating without mutating

Write `with_timeout(config, seconds)` that returns a configuration like `config` but with `config["server"]["timeout"]` changed, without changing `config` and without `deepcopy`.

SOLUTION

```
def with_timeout(config, seconds):
    return {**config, "server": {**config["server"], "timeout": seconds}}

base = {"name": "api", "server": {"host": "0.0.0.0", "timeout": 30, "tags": ["prod"]}}
fast = with_timeout(base, 5)
assert base["server"]["timeout"] == 30 and fast["server"]["timeout"] == 5
assert fast["tags"] is base["tags"]           # untouched parts are shared, safely
```

`{**config, ...}` builds a new dict with all of `config`'s entries, then overrides `"server"` with a new dict built the same way. Only the path to the change is copied: two small dicts. Everything else, like the `tags` list, is shared between the old and new configurations, which is safe as long as nobody mutates it. That's the same structural sharing as

chapter 19 of *The Engineer's Press*'s persistent lists. `deepcopy` would also work but copies everything, and copies things it shouldn't if the config holds, say, a logger.

EXERCISE 15.3 The cache that ate the memory

A web service memoises an expensive function with a module-level dict, `_cache[key] = result`, keyed by request parameters. Its memory grows until it's restarted. Why, and what are two fixes?

SOLUTION

The dict is a global, so it lives as long as the process, and every entry it holds keeps its result alive. Nothing ever removes an entry, so with a steady stream of new parameters it grows without limit; Python can't free the results because the program is still, technically, referring to them. The first fix is a bounded cache:

`@functools.lru_cache(maxsize=10_000)` keeps the 10,000 most recently used results and discards the rest, so memory has a ceiling. The second is expiry: store a timestamp with each entry and remove entries older than some age, which also handles results that go stale. Which is right depends on whether the results can change. Either way, the lesson from the reference-counting section applies: garbage collection frees what's unreachable, and a cache is the opposite of unreachable.

CHECKLIST

- ☐ Can you explain what frames hold, what lives on the heap, and what a reference is?
- ☐ Can you explain call by sharing, and predict whether a function's change is visible to its caller?
- ☐ Can you spot the grid, `fromkeys`, default-argument and class-attribute aliasing bugs?
- ☐ Can you choose between a shallow copy, a deep copy, and building a new value?
- ☐ Can you explain reference counting, cycles, and how a Python program leaks memory?

GO DEEPER

COURSE CS50x, week 4: Memory David J. Malan, Harvard

DOCS copy: Shallow and deep copy operations Python documentation

TALK Python Names and Values (PyCon 2015) Ned Batchelder

Modules, packages and environments

3 exercises

A program of any size is split across files, and most programs also use code that other people wrote. Python's import system finds and loads your files, packages group them, and a virtual environment gives each project its own set of installed libraries so that one project's upgrade can't break another. On top of that, a dependency resolver has to choose versions that satisfy every library's requirements at once, and a lockfile records the choice so the program installs identically everywhere. This chapter builds a small package, runs into a circular import, creates an environment, and resolves a set of dependencies by search.

Modules and import

A **module** is a Python file, and `import shop` makes its contents available as `shop.something`. The first time a module is imported, Python finds the file, runs it from top to bottom, and stores the resulting module object in `sys.modules`. Every later import of the same module, from anywhere in the program, gets that stored object without running the file again, as the example shows: the second import printed nothing. So a module's top-level code runs once per program, which is why it should mostly define things, functions, classes and constants, rather than do things.

Python looks for modules in the folders listed in `sys.path`, in order, and the first entry is the folder of the script being run. That has a famous consequence: a script named `random.py` or `json.py` in your project hides the standard library module of the same name, for every program run from that folder, and the error it produces is baffling. Name your files after what they do, not after modules you use.

`from shop.prices import price` imports one name into the current module; `import shop.prices` keeps the module's name as a prefix. Prefer the second, or `from` imports of a few specific names. Avoid `from module import *`, which dumps every name into your module and makes it impossible to tell where a name came from.

EXAMPLE

`modules.py` · a package imported once and cached, run as a program with `-m`, a circular import, a virtual environment, and dependency resolution with backtracking

```
import hashlib, pathlib, subprocess, sys, tempfile, textwrap, venv

files = {
    "shop/__init__.py": 'print("    (running shop/__init__.py)")\n',
    "shop/prices.py": textwrap.dedent('''\
print("    (running shop/prices.py)")
PRICES = {"tea": 2.5, "cake": 3.0}
```

```

    def price(item): return PRICES[item]
    '''),
    "shop/cart.py": textwrap.dedent('''\
    from .prices import price          # a relative import, inside the package
    def total(items): return sum(price(i) for i in items)
    if __name__ == "__main__":        # true only when run as a program
        print("    cart run as a program: total(['tea', 'cake']) =", total(["tea", "cake"]))
    '''),
    "loop_a.py": "from loop_b import helper\ndef main(): return helper()\n",
    "loop_b.py": "from loop_a import main\ndef helper(): return 42\n",
}

with tempfile.TemporaryDirectory() as root:
    for name, text in files.items():
        path = pathlib.Path(root, name); path.parent.mkdir(exist_ok=True);
        path.write_text(text)
    sys.path.insert(0, root)                                # where import looks, first

    # 1. A module's code runs once, on first import; later imports reuse it from sys.modules
    print("import shop.cart:")
    import shop.cart
    print("import shop.prices again:")
    import shop.prices                                     # nothing runs: it's already in sys.modules
    print(f"shop.cart.total(['tea', 'tea']) = {shop.cart.total(['tea', 'tea'])}; "
          f"{'shop.prices' in sys.modules: {'shop.prices' in sys.modules}}")

    # 2. The same file as a program: python -m runs it with __name__ set to "__main__"
    run = subprocess.run([sys.executable, "-m", "shop.cart"], cwd=root, capture_output=True,
                          text=True)
    print(f"\npython -m shop.cart:\n{run.stdout}", end="")

    # 3. Two modules that import from each other at the top fail, depending on which goes first
    try: import loop_a
    except ImportError as e: print(f"\nimport loop_a -> ImportError: {e}".replace(root + "/",
    ""))

    # 4. A virtual environment: its own interpreter and its own site-packages
    env = pathlib.Path(root, "env")
    venv.create(env, with_pip=False)
    python = env / ("Scripts/python.exe" if sys.platform == "win32" else "bin/python")
    check = "import sys; print(sys.prefix != sys.base_prefix, sys.prefix.endswith('env'))"
    out = subprocess.run([str(python), "-c", check], capture_output=True,
                          text=True).stdout.split()
    print(f"\ninside the new environment: in a venv {out[0]}, prefix is the env folder {out[1]}")
    sys.path.remove(root)

    # 5. Dependency resolution: pick one version of everything so that every constraint holds
    INDEX = {                                              # package -> version -> requirements (lowest,
    below)
    "app": {(1, 0): {"web": ((2, 0), (3, 0)), "orm": ((1, 0), (2, 0))}},
    "web": {(2, 1): {"util": ((1, 0), (2, 0))}, (2, 4): {"util": ((1, 5), (2, 0))}, (3, 0):
    {}},
    "orm": {(1, 2): {"util": ((1, 0), (1, 9))}, (1, 3): {"util": ((1, 2), (1, 5))}},
    "util": {(1, 1): {}, (1, 4): {}, (1, 6): {}, (1, 8): {}},
    }

    show = lambda v: ".".join(map(str, v))
    def resolve(wanted, chosen, depth=0):
        """Depth-first, newest version first, backing up when a later constraint can't be met."""
        if not wanted: return chosen
        (name, (lo, hi)), rest = wanted[0], wanted[1:]
        pad = " " * depth
        if name in chosen:                                # already picked: it must satisfy this range
            too
            if lo <= chosen[name] < hi: return resolve(rest, chosen, depth)
            print(f"{pad}conflict: {name} {show(chosen[name])} isn't >={show(lo)},<{show(hi)}")
            return None

```

```

for version in sorted(INDEX[name], reverse=True):
    if lo <= version < hi:
        print(f"{pad}try {name} {show(version)}")
        needs = list(INDEX[name][version].items())
        found = resolve(rest + needs, {**chosen, name: version}, depth + 1)
        if found: return found
    return None
print("\nresolving app 1.x:")
lock = resolve(["app", ((1, 0), (2, 0))], {})
print("the lockfile pins exact versions, each with the hash of its file (here, a stand-in):")
for name, version in sorted(lock.items()):
    digest = hashlib.sha256(f"{name}-{show(version)}".encode()).hexdigest()[:16]
    print(f"    {name} + '==' + show(version):<10} --hash=sha256:{digest}...")

```

OUTPUT

```

import shop.cart:
  (running shop/__init__.py)
  (running shop/prices.py)
import shop.prices again:
shop.cart.total(['tea', 'tea']) = 5.0; 'shop.prices' in sys.modules: True

python -m shop.cart:
  (running shop/__init__.py)
  (running shop/prices.py)
  cart run as a program: total(['tea', 'cake']) = 5.5

import loop_a -> ImportError: cannot import name 'main' from partially initialized module
  'loop_a' (most likely due to a circular import) (loop_a.py)

inside the new environment: in a venv True, prefix is the env folder True

resolving app 1.x:
  try app 1.0
    try web 2.4
      try orm 1.3
        try util 1.8
          conflict: util 1.8 isn't >=1.2,<1.5
        try util 1.6
          conflict: util 1.6 isn't >=1.2,<1.5
      try orm 1.2
        try util 1.8
the lockfile pins exact versions, each with the hash of its file (here, a stand-in):
  app==1.0 --hash=sha256:41fd04e97ae65f82...
  orm==1.2 --hash=sha256:d05c9cd285d070a1...
  util==1.8 --hash=sha256:acafc630cc5badbd...
  web==2.4 --hash=sha256:e2b26c201e4c01a1...

```

Packages and running as a program

A **package** is a folder of modules with an `__init__.py` file, which runs when the package is first imported and can be empty. Inside a package, modules can import each other with **relative imports**, `from .prices import price`, where the dot means "this package", so the package keeps working if it's renamed or moved. Every module has a `__name__`: its module name when imported, and `"__main__"` when run as the program. The idiom `if __name__ == "__main__":` guards code that should run only when the file is used as a program, which lets the same file be both a library and a command. Run a module inside a package with `python -m shop.cart`, which sets up the import path correctly, rather than `python shop/cart.py`, which breaks relative imports.

Circular imports

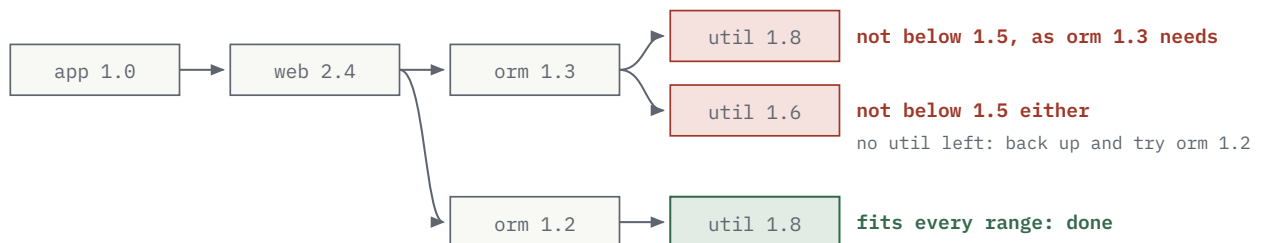
The example's `loop_a` imports a name from `loop_b`, and `loop_b` imports a name from `loop_a`. Importing `loop_a` starts running it, and its first line starts importing `loop_b`, whose first line asks for `loop_a.main`. But `loop_a` is only partly run: it hasn't reached the `def main` line yet. Python reports exactly that, "cannot import name 'main' from partially initialized module 'loop_a' (most likely due to a circular import)". The cure is structural. Two modules that need each other usually contain a third concept that both depend on, which belongs in its own module; failing that, import inside the function that needs the name, so the import happens when the function runs, after both modules are loaded.

Environments, dependencies and lockfiles

Third-party libraries come from the Python Package Index, PyPI, and are installed with `pip` or a newer tool such as `uv`. Installing them into your one system Python mixes every project's libraries together, so upgrading a library for one project can break another, and nobody can tell which project needs what. A **virtual environment** solves this: a folder with its own Python interpreter and its own `site-packages`, created with `python -m venv .venv` and activated with `source .venv/bin/activate`, or `.venv\Scripts\activate` on Windows. Inside it, `sys.prefix` points at the environment, which is how the example's check confirms it's running in one. Make one per project, and never install project libraries globally.

A project declares what it needs in `pyproject.toml`, usually as ranges: `requests >= 2.31, < 3`. Libraries declare ranges too, and their requirements have requirements. Choosing one version of every package so that all the ranges hold at once is **dependency resolution**, and in general it's a search problem. The example's resolver tries the newest allowed version of each package first and backs up when a later constraint fails: `web 2.4` needs `util >= 1.5` but `orm 1.3` needs `util < 1.5`, so no `util` fits, and it retreats to `orm 1.2`, which accepts `util 1.8`. `pip`'s resolver works in the same spirit, backtracking through versions, at a much larger scale.

SEARCHING FOR VERSIONS THAT SATISFY EVERY CONSTRAINT



web 2.4 needs `util >= 1.5`; orm 1.3 needs `util < 1.5`; orm 1.2 accepts anything below 1.9
the lockfile then records exactly app 1.0, web 2.4, orm 1.2 and util 1.8

Dependency resolution as a search with backtracking. Each choice narrows what's allowed later, and a dead end sends the search back to the most recent choice that still has alternatives.

Because ranges allow many answers, the resolver's choice can change from one day to the next as new versions are published, and "it works on my machine" follows. A **lockfile** records the result: the exact version of every package, direct and indirect, with a hash of each package file, so that installing from the lockfile reproduces the environment exactly and a tampered download is rejected. Tools such as uv, Poetry and pip-tools generate them. The rule of thumb: applications commit a lockfile and install from it; libraries publish ranges, so that the applications using them can resolve compatible versions.

COMMON TRAPS

Naming your file after a module you import, such as `random.py`, so it hides the real one.

Doing work at import time, such as connecting to a database, so importing the module has side effects.

Installing project libraries into the system Python instead of a virtual environment.

Unpinned dependencies in an application, so a new release of a library breaks a deploy.

Two modules importing each other at the top, and patching the symptom instead of moving the shared part out.

EXERCISES

EXERCISE 16.1 A file called `random.py`

A student writes a dice game in `random.py` that starts with `import random` and calls `random.randint(1, 6)`. On Python 3.12 it fails with `AttributeError: partially initialized module 'random' has no attribute 'randint' (most likely due to a circular import)`. Explain each part of the message, and what newer versions say instead.

SOLUTION

The script's own folder is first in `sys.path`, so `import random` finds the student's `random.py`, not the standard library's. A script being run is registered as `__main__`, not `random`, so importing it under the name `random` starts running the same file a second time, as a new module. That second run's first line is `import random` again, and this time Python finds the module it's in the middle of loading in `sys.modules` and returns it: the "circular import". Execution then reaches `random.randint` while that module contains only what has run so far, which doesn't include any `randint`: "partially initialized module has no attribute". Python 3.13 and later recognise the situation and say so

directly: "module 'random' has no attribute 'randint' (consider renaming '.../random.py' since it has the same name as the standard library module named 'random!...)". The fix is to rename the file, to `dice.py` say, and delete any `__pycache__/random.*.pyc` it left behind.

EXERCISE 16.2 Breaking the cycle

Fix the example's `loop_a` and `loop_b` two ways: by moving code, and by changing where an import happens. Which is better?

SOLUTION

Moving code: `loop_b` only imports `main` from `loop_a` and doesn't use it, and `loop_a` needs `helper`. If `loop_b` genuinely needed something from `loop_a`, the shared piece would move to a third module, `common.py`, imported by both, and neither would import the other. Changing where the import happens: in `loop_b`, replace the top-level import with one inside the function that uses `main`, `def run(): from loop_a import main; return main()`, so it executes only when `run` is called, after both modules have finished loading. The first is better. A cycle usually means two modules are really one concept split badly, or share a concept that deserves its own home, and fixing the structure removes the problem permanently. The deferred import works but hides the dependency, and is best kept for breaking cycles in code you can't restructure.

EXERCISE 16.3 Ranges and pins

You maintain both a library, `tinyhttp`, and an application that uses it. Which of them should declare `certifi >= 2024.2`, and which should commit a lockfile with `certifi==2026.4.12`? What goes wrong if they swap?

SOLUTION

The library declares a range, and the application commits the lockfile. A library is installed alongside other libraries that have their own requirements, so it should accept every version it actually works with; a range leaves the resolver room to find one version that satisfies everyone. The application is the final product, and its goal is that every install, on every machine and in production, gets exactly the same, tested set of packages, so it pins everything, including indirect dependencies, with hashes. If the library pinned `certifi==2026.4.12`, any application also using a library that needs a different `certifi` version would be impossible to resolve. If the application used only

ranges, two deploys a week apart could install different versions, and a bug introduced by an upstream release would appear without any change to the application's code. The application still upgrades: deliberately, by regenerating the lockfile, running the tests, and committing the change.

CHECKLIST

- ☐ Can you explain what happens on the first and the second import of a module?
- ☐ Can you build a package with relative imports and run a module in it with `python -m`?
- ☐ Can you diagnose and fix a circular import, and a module shadowing the standard library?
- ☐ Can you create and use a virtual environment per project?
- ☐ Can you explain dependency resolution, and why applications use lockfiles while libraries use ranges?

GO DEEPER

DOCS [The Python Tutorial: Modules](#) Python documentation

GUIDE [Python Packaging User Guide](#) Python Packaging Authority

DOCS [uv: an extremely fast Python package and project manager](#) Astral

PART V

Programs in the world

Four chapters on programs that other people use and other people wrote. Most of your working life will be spent in code you didn't write, so the part begins with reading unfamiliar code and changing it without breaking it. Then three of the stage's deliverables, each in a small working form: a command-line tool that behaves like one, a game whose state is a value you can save, replay and undo, and a client for a network API that copes with slow, failing and rate-limited servers, and caches what it learns.

Reading unfamiliar code

3 exercises

Most code you'll work on was written by someone else, often someone who has left, and changing it starts with understanding it well enough not to break it. Reading code is a skill with techniques of its own: map the structure before the details, find out what touches shared state, and use the running program to answer questions the text can't. Changing it safely has techniques too: record what the code does now, for many inputs, before you change a line. This chapter maps a small legacy module automatically, pins down its behaviour with 686 recorded answers, and shows those answers catching a refactoring mistake that looked harmless.

Map before you read

Reading a file top to bottom, line by line, is the slowest way to understand it. Start with the shape. What does the module define, and what's at module level? Which functions call which? Which ones read or change state that lives outside them, globals, attributes, files? Those last are where surprises come from, because their behaviour depends on more than their arguments. Then find the entry points, where the program starts or where other code calls in, and the tests, if there are any, which are often the best documentation of what the code is supposed to do.

Editors answer most of these questions directly: "go to definition", "find all references" and a call hierarchy view are worth learning thoroughly. On the command line, `grep -rn "calc(" .`, or the faster `rg`, finds every use of a name. `git log -p file.py` and `git blame` tell you when and why each line changed, which explains code that looks wrong but isn't. The example does the mapping with Python's `ast` module, which parses code into the same tree Python itself uses, so a dozen lines can list every function, what it calls, which module-level names it reads, and which it mutates.

EXAMPLE

reading.py · mapping a legacy module with ast, recording 686 characterisation outputs, and checking a careful and a careless refactor against them

```
import ast, itertools

LEGACY = '''
RATE = 0.2
log = []
def calc(items, cust):
    t = 0
    for i in items:
        p = i[1] * i[2]
        if i[0] in SPECIAL: p = p * 0.9
        t = t + p
```

```

        if cust.get("vip"): t = t - disc(t)
        t = round(t, 2)
        log.append((cust["name"], t))
        return round(t + tax(t), 2)
def disc(t):
    if t > 100: return 10
    return t * 0.05
def tax(t): return round(t * RATE, 2)
SPECIAL = {"tea"}
'''

# 1. Before changing code, map it: what's defined, what calls what, what touches shared state
tree = ast.parse(LEGACY)
top_names = {t.id for node in tree.body if isinstance(node, ast.Assign) for t in node.targets}
print(f"module-level names: {sorted(top_names)}")
for fn in (n for n in tree.body if isinstance(n, ast.FunctionDef)):
    calls = sorted({c.func.id for c in ast.walk(fn)
                    if isinstance(c, ast.Call) and isinstance(c.func, ast.Name)})
    used = sorted({n.id for n in ast.walk(fn) if isinstance(n, ast.Name)} & top_names)
    mutated = sorted({c.func.value.id for c in ast.walk(fn) if isinstance(c, ast.Call)
                      and isinstance(c.func, ast.Attribute) and isinstance(c.func.value,
                      ast.Name)
                      and c.func.value.id in top_names})
    lines = fn.end_lineno - fn.lineno + 1
    size = f"{lines} line{'s' if lines > 1 else ''}"
    print(f"    {fn.name}({'', '.join(a.arg for a in fn.args.args)}): {size}, calls {calls}, "
          f"reads globals {used}, mutates {mutated or 'nothing'}")

# 2. Pin down what it does now, for many inputs: characterisation tests
def load(source):
    space = {}; exec(source, space); return space
legacy = load(LEGACY)
products = [("tea", 3.0), ("cake", 4.5), ("jam", 6.25)]
cases = [([(name, price, qty) for (name, price), qty in zip(products, qtys)], {"name": "c",
    "vip": vip})
          for qtys in itertools.product(range(0, 25, 4), repeat=3) for vip in (False, True)]
golden = [legacy["calc"](items, cust) for items, cust in cases]
print(f"\nrecorded {len(golden)} outputs from the legacy calc, e.g. {golden[100]} and "
      {golden[-1]}")

# 3. Now refactor, and check every recorded case still gives the same answer
REFACTORED = '''
TAX_RATE = 0.2
SPECIAL_OFFERS = {"tea"}
SPECIAL_DISCOUNT = 0.9
def line_total(name, price, qty):
    total = price * qty
    return total * SPECIAL_DISCOUNT if name in SPECIAL_OFFERS else total
def vip_discount(subtotal):
    return 10 if subtotal > 100 else subtotal * 0.05
def tax(amount):
    return round(amount * TAX_RATE, 2)
def calc(items, customer, history=None):
    subtotal = sum(line_total(*item) for item in items)
    if customer.get("vip"): subtotal -= vip_discount(subtotal)
    subtotal = round(subtotal, 2)
    if history is not None: history.append((customer["name"], subtotal))
    return round(subtotal + tax(subtotal), 2)
'''
CARELESS = REFACTORED.replace("subtotal > 100", "subtotal >= 100") # "it's the same,
really"
for label, source in (("careful refactor", REFACTORED), ("careless refactor", CARELESS)):
    calc = load(source)["calc"]
    diffs = [(i, g, calc(*cases[i])) for i, g in enumerate(golden) if calc(*cases[i]) != g]
    if not diffs: verdict = "identical on every case"
    else:

```

```
i, was, now = diffs[0]
verdict = f"{len(diffs)} of {len(golden)} differ, e.g. case {i}: {was} became {now}"
print(f"{label}: {verdict}")
```

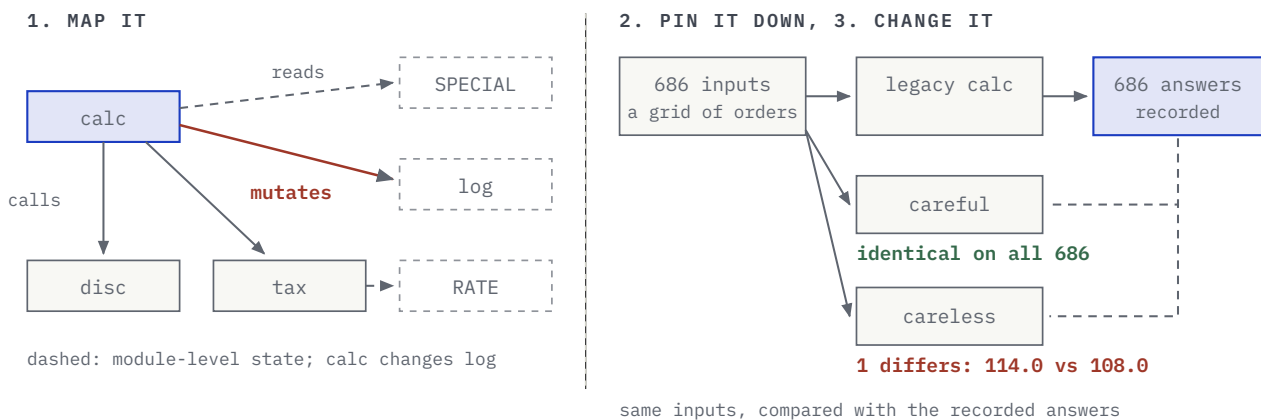
OUTPUT

```
module-level names: ['RATE', 'SPECIAL', 'log']
calc(items, cust): 10 lines, calls ['disc', 'round', 'tax'], reads globals ['SPECIAL',
'log'], mutates ['log']
disc(t): 3 lines, calls [], reads globals [], mutates nothing
tax(t): 1 line, calls ['round'], reads globals ['RATE'], mutates nothing
```

```
recorded 686 outputs from the legacy calc, e.g. 42.96 and 375.36
careful refactor: identical on every case
careless refactor: 1 of 686 differ, e.g. case 9: 114.0 became 108.0
```

What the map says

The legacy module is short, and still the map tells you things a quick read misses. `calc` calls `disc` and `tax`, reads `SPECIAL`, and **mutates** the global `log`: every call appends to it, so calling `calc` twice is not the same as calling it once, and any test that calls it leaves a trace for the next. `SPECIAL` is defined at the bottom of the file, after the function that uses it, which works only because it's looked up when `calc` runs, not when it's defined. And the names `t`, `p` and `i` carry no meaning, so the purpose of each line has to be reconstructed from arithmetic. None of this is a bug yet. All of it is a place where a change can go wrong.



Understand, then pin down, then change. The map shows where shared state is touched; the recorded answers turn "I think it still works" into a check that runs in a second.

Characterisation tests

Before changing code that has no tests, write tests that record what it does now, right or wrong. Michael Feathers, who named them **characterisation tests** in *Working Effectively with Legacy Code*, put the idea plainly: legacy code is code without tests, and the first job is to get it under test. The example runs `calc` on a grid of 686 orders, every combination of quantities for three products, with and without VIP status, and stores the answers. They

aren't claims that the answers are correct, only that they're what the code produces, which is what existing users depend on. If a change makes any answer differ, you've changed behaviour, and you must decide whether you meant to.

The careful refactor renames everything, extracts `line_total` and `vip_discount`, names the magic numbers, and replaces the global `log` with an optional `history` parameter. Every one of the 686 answers is identical. (The `history` change does alter behaviour on purpose: callers that relied on the global `log` must now pass a list. A deliberate interface change like that belongs in its own step, announced to callers, not mixed in with a refactor.) The careless refactor changes one character, `>` to `>=`, which looks like a harmless tidy-up. One answer in 686 differs: a VIP order of exactly £100 now gets the £10 discount instead of 5%. No reviewer would reliably spot that; the recorded answers did in under a second.

Changing code in small steps

Refactoring means changing a program's structure without changing its behaviour, and it's done in small steps, each followed by running the tests: rename a variable, run; extract a function, run; replace a magic number with a named constant, run. If a step breaks something, the cause is the last small change, not one of fifty. Keep refactoring and behaviour changes in separate commits, so a reviewer can check the first for "nothing changed" and the second for "the right thing changed". Commit after each working step, so you can always go back. The habit feels slow and is the fastest way there is to change code you don't fully understand.

COMMON TRAPS

Reading line by line from the top instead of mapping structure and state first.

Refactoring without tests, and relying on careful reading to spot behaviour changes.

Mixing a refactor with a bug fix or feature in one change, so neither can be reviewed.

"Fixing" code that looks wrong before finding out, with `git blame` and the tests, why it's like that.

Characterisation tests with too few inputs to reach the boundaries where behaviour changes.

EXERCISES

EXERCISE 17.1 The side effect the tests ignore

The example's characterisation tests compare only return values, but the legacy `calc` also appends to `log`. Extend the tests to cover that side effect, and show they'd catch a refactor that logged the total after tax instead of before.

SOLUTION

```
legacy = load(LEGACY)
golden = []
for items, cust in cases:
    legacy["log"].clear()
    total = legacy["calc"](items, cust)
    golden.append((total, list(legacy["log"])))

calc = load(REFACTORED)["calc"]
for (items, cust), (total, logged) in zip(cases, golden):
    history = []
    assert (calc(items, cust, history), history) == (total, logged)
```

Record what the code did to the outside world along with what it returned: here, the entries appended to `log` during each call, clearing it first so each case is independent. The refactored version is checked by passing a fresh list as `history` and comparing both parts. A refactor that appended the after-tax total leaves every return value the same, and fails 684 of the 686 cases on the log, all but the two empty orders, where tax is zero: exactly the kind of change return-value tests can't see. In general a characterisation test must capture every effect a caller could depend on: return values, exceptions raised, files written, messages sent, state changed.

EXERCISE 17.2 A grid that reaches the boundary

The careless refactor was caught only because one order in the grid happened to total exactly £100. How would you design the inputs so boundaries like that are covered on purpose?

SOLUTION

Read the code for its constants and comparisons, then make sure inputs land on each side of each one. `disc` compares with 100, so add orders whose subtotal is 99.99, exactly 100 and 100.01, each with and without VIP; `tax` rounds to pennies, so add subtotals whose tax falls exactly on a half-penny; the special offer applies to tea, so include orders with and without it, and with only tea. Zero quantities, a single item and large orders cover the other edges. Then add a few hundred random orders, seeded, for what you didn't think of, as chapter 11 recommends. The example's grid reached 100 by luck: 16 jams at £6.25.

A grid with different prices could have missed it, and the careless refactor would have passed, which is the strongest argument for choosing boundary inputs deliberately.

EXERCISE 17.3 Why is this here?

You find `time.sleep(0.2)` inside a loop that uploads files, with no comment. It makes the job slow, and removing it looks safe. What do you do before deleting it?

SOLUTION

Find out why it was added. `git blame` on that line gives the commit, and the commit message, the pull request it came from and any linked issue usually explain it, for example "the storage service rate-limits us to 5 requests a second; without this we get 429 errors". Search the code and issue tracker for the same number, and ask whoever wrote or reviewed it if they're around. If the reason still holds, the fix isn't deleting the sleep but replacing it with something better, such as honouring the server's `Retry-After` header, chapter 20, and adding the comment that should have been there. If the reason no longer holds, delete it, and say in the commit message why it's now safe. This is Chesterton's fence: don't remove a fence until you know why it was built.

CHECKLIST

- ☐ Can you map an unfamiliar module's functions, calls and shared state before reading the details?
- ☐ Can you use go-to-definition, find-references, `grep` and `git blame` to answer questions about code?
- ☐ Can you write characterisation tests that capture return values and side effects?
- ☐ Can you refactor in small steps, running the tests after each?
- ☐ Can you keep refactors and behaviour changes in separate, reviewable commits?

GO DEEPER

BOOK Working Effectively with Legacy Code Michael Feathers

SITE Catalog of Refactorings Martin Fowler

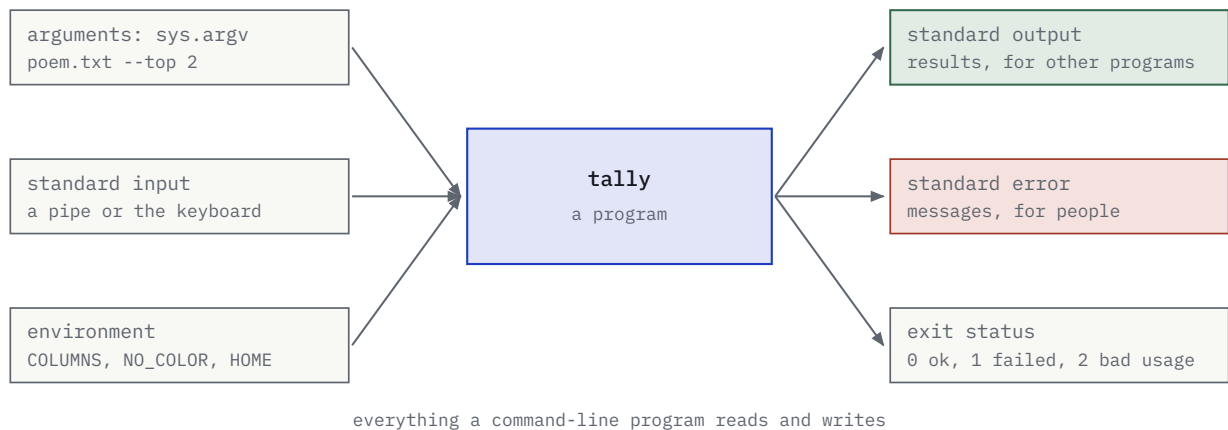
Command-line tools

3 exercises

A command-line program is used by people at a terminal and, just as often, by other programs: scripts, scheduled jobs, pipelines. Both depend on the same conventions. Arguments and a help screen, results on standard output and errors on standard error, standard input when no file is named, and an exit status that says whether it worked. A tool that follows them composes with every other tool; one that doesn't is a dead end. This chapter writes a word-counting tool with `argparse` and runs it as a real program, so you can see its help screen, its output, its errors and its exit statuses.

What a program reads and writes

A command-line program has three inputs and three outputs. Its **arguments** arrive as a list of strings, `sys.argv`, split by the shell. **Standard input** is a stream that a pipe, a redirected file or the keyboard can supply. The **environment** holds settings such as `HOME`, `COLUMNS` and `NO_COLOR`. On the way out, **standard output** carries results, **standard error** carries messages for people, and the **exit status**, a small number, tells the caller whether it worked: 0 for success, anything else for failure. Chapter 3 of *The Engineer's Loom* shows how a shell wires these together; here the point is to use each for its job.



Everything a command-line program reads and writes. Keeping results on standard output and messages on standard error is what lets another program use the results.

The separation between the two output streams matters more than it looks. `tally *.txt | sort -n` sorts tally's results; if tally printed a warning to standard output, the warning would be sorted in with them. Written to standard error, it still appears on the terminal and

stays out of the data. Likewise, exit statuses are how scripts decide what to do next: `tally notes.txt && echo done` prints "done" only if tally succeeded, and a scheduled job is marked failed when its status isn't 0.

Arguments with argparse

Parsing `sys.argv` by hand is tedious and always slightly wrong. The standard library's `argparse` does it properly. You declare each argument: a positional `files` taking any number of values, `nargs="*"`; an option `-t` or `--top` with `type=int` so it's converted and checked; a `--version` action. From that it generates the usage line and the `--help` screen, converts types, and rejects bad input with a usage message on standard error and exit status 2, as the example's `--top lots` shows, before any of your code runs. Add a sentence of help to every argument; the help screen is the first documentation most users read.

EXAMPLE

`cli.py` · a word-counting tool built with `argparse` and run as a real program, showing its help, output, standard input, an error on standard error, a usage error, and exit statuses

```
import os, pathlib, subprocess, sys, tempfile, textwrap

TOOL = textwrap.dedent('''\
    """tally: count lines, words and the commonest words in text files or standard input."""
    import argparse, collections, re, sys

    def count(text):
        words = re.findall(r"[a-z']+", text.lower())
        return text.count("\n"), len(words), collections.Counter(words)

    def main(argv=None):
        parser = argparse.ArgumentParser(prog="tally", description=__doc__.split(": ", 1)[1])
        parser.add_argument("files", nargs="*", help="files to read (default: standard input)")
        parser.add_argument("-t", "--top", type=int, default=0, metavar="N",
                            help="also list the N commonest words")
        parser.add_argument("--version", action="version", version="tally 1.0")
        args = parser.parse_args(argv)          # bad arguments: message and exit
        status = 2
        if args.top < 0: parser.error("--top must be 0 or more")
        sources = args.files or ["-"]
        status = 0
        for name in sources:
            try:
                text = sys.stdin.read() if name == "-" else open(name, encoding="utf-8").read()
            except OSError as e:
                print(f"tally: {name}: {e.strerror}", file=sys.stderr)    # errors go to stderr
                status = 1
                continue
            lines, words, common = count(text)
            print(f"{lines:>5} {words:>6} {'(stdin)' if name == '-' else name}")    # like wc
            for word, n in common.most_common(args.top):
                print(f"        {n:>4} {word}")
        return status                                # 0 means success to the shell

if __name__ == "__main__":
    sys.exit(main())
''')
```

```

with tempfile.TemporaryDirectory() as folder:
    tool = pathlib.Path(folder, "tally.py"); tool.write_text(TOOL)
    pathlib.Path(folder, "poem.txt").write_text("the rain in spain\nfalls mainly\non the
    plain\n")
    env = {**os.environ, "COLUMNS": "80", "NO_COLOR": "1"} # fixed width, plain text help
    def shell(*args, stdin=None):
        r = subprocess.run([sys.executable, str(tool), *args], input=stdin,
                           capture_output=True,
                           text=True, cwd=folder, env=env)
        print(f"$ tally {' '.join(args)}.rstrip() + (f" (input: {stdin!r})" if stdin else
        ""))
        for line in (r.stdout + r.stderr).rstrip().splitlines(): print((" " + line).rstrip())
        print(f" [exit status {r.returncode}]\n")
    shell("--help")
    shell("poem.txt", "--top", "2")
    shell(stdin="to be or not to be\n")
    shell("poem.txt", "missing.txt")
    shell("--top", "lots")
    shell("--version")

```

OUTPUT

```

$ tally --help
usage: tally [-h] [-t N] [--version] [files ...]

count lines, words and the commonest words in text files or standard input.

positional arguments:
  files                files to read (default: standard input)

options:
  -h, --help          show this help message and exit
  -t, --top N         also list the N commonest words
  --version           show program's version number and exit
[exit status 0]

$ tally poem.txt --top 2
   3      9 poem.txt
   2 the
   1 rain
[exit status 0]

$ tally (input: 'to be or not to be\n')
   1      6 (stdin)
[exit status 0]

$ tally poem.txt missing.txt
   3      9 poem.txt
tally: missing.txt: No such file or directory
[exit status 1]

$ tally --top lots
usage: tally [-h] [-t N] [--version] [files ...]
tally: error: argument -t/--top: invalid int value: 'lots'
[exit status 2]

$ tally --version
tally 1.0
[exit status 0]

```

The conventions that make it composable

- **Read standard input when no file is given**, and treat `-` as standard input. Then tally works at the end of a pipeline, `curl -s https://example.com/page | tally --top 10`, as well as on files.
- **Carry on past a bad file when it makes sense**, report it on standard error, and exit non-zero at the end. The example counts `poem.txt`, reports `missing.txt` and returns 1, so the user gets the useful results and the script running it learns that something failed.
- **Use exit status 0 for success, 1 for failure, 2 for bad usage**. `argparse` uses 2 already; shells report 128 plus the signal number for a program killed by a signal, such as 130 after Ctrl-C.
- **Print results as plain, regular lines**, one record per line with consistent columns, so `sort`, `grep`, `awk` and spreadsheets can use them. If you also offer a human-friendly format, make machine-readable output, such as `--json`, available too.
- **Never prompt when standard input isn't a terminal**, and respect `NO_COLOR`: a program run from a script has nobody to answer questions, and colour codes in a log file are noise.

Structure for testing and installing

The example's tool keeps all its logic in `main(argv=None)`, which parses `argv` if given, or the real command line if not, and returns the exit status; the last lines, `if __name__ == "__main__": sys.exit(main())`, are the only part that runs when it's a program. That shape makes the tool testable without starting a process: a test calls `main(["poem.txt", "--top", "2"])` and checks the return value, and pytest's `capsys` fixture captures what it printed. The example starts a real process with `subprocess.run` instead, which is slower but tests exactly what a user gets, including the exit status the shell sees.

To install the tool as a command, declare it in `pyproject.toml` under `[project.scripts]` as `tally = "tally:main"`; installing the package, for instance with `uv tool install .` or `pipx install .`, then puts a `tally` command on the path that calls `main`. For tools with many subcommands, like `git`, `argparse` supports them through `add_subparsers`, and third-party libraries such as Click and Typer build the same interfaces from decorators or type hints.

COMMON TRAPS

Printing errors to standard output, where they corrupt the results another program is reading.

Exiting 0 after a failure, so scripts and schedulers think everything worked.

Requiring a file argument instead of reading standard input, so the tool can't be used in a pipe.

Parsing `sys.argv` by hand, and getting quoting, missing values or `--help` wrong.

Interactive prompts and colour codes when the output isn't a terminal

EXERCISES

EXERCISE 18.1 A new option

Add `--min-length N` to tally, so the commonest-words list ignores words shorter than `N` characters. Reject negative values with a proper usage error, and write a test that calls `main` directly.

SOLUTION

```
parser.add_argument("--min-length", type=int, default=1, metavar="N",
                    help="ignore words shorter than N in the --top list")
...
if args.min_length < 0: parser.error("--min-length must be 0 or more")
...
common = collections.Counter({w: n for w, n in common.items() if len(w) >=
                               args.min_length})

def test_min_length_filters_short_words(tmp_path, capsys):
    f = tmp_path / "t.txt"; f.write_text("a a a bb bb ccc\n")
    assert main([str(f), "--top", "1", "--min-length", "2"]) == 0
    assert capsys.readouterr().out.splitlines()[-1].split() == ["2", "bb"]
```

`type=int` makes `argparse` reject non-numbers with exit status 2, and `parser.error` does the same for values that are numbers but make no sense, with the same usage message. The test uses `pytest`'s `tmp_path` for a real file and `capsys` to capture standard output, and calls `main` in-process, which is fast. The default of 1 keeps the old behaviour for everyone who doesn't use the option, which is the rule for adding options to a tool people already depend on.

EXERCISE 18.2 Exit statuses in a script

A nightly job runs `tally /data/*.txt > counts.txt`, then uploads `counts.txt`. One night a file is unreadable. What does the job do with the example's tally, and what would happen if tally had exited 0 anyway?

SOLUTION

Tally counts every readable file, writes those results to `counts.txt`, prints `tally: /data/x.txt: Permission denied` to standard error, which goes to the job's log, not the file, and exits 1. A job script written with `set -e`, or `tally ... > counts.txt || exit 1`, stops before uploading incomplete data, and the scheduler marks the job failed and alerts someone. Had tally exited 0, the job would upload a file missing

one input's counts, and nothing anywhere would record that it was incomplete until someone noticed the numbers were low, perhaps weeks later. The exit status is the only part of a program's output that every caller checks without parsing anything, which is why getting it right matters more than the wording of any message.

EXERCISE 18.3 Subcommands

Sketch an argparse setup for a notes tool with three subcommands:

`notes add TEXT`, `notes list [--tag TAG]` and `notes delete ID`, where `ID` must be an integer.

SOLUTION

```
parser = argparse.ArgumentParser(prog="notes")
sub = parser.add_subparsers(dest="command", required=True)
add = sub.add_parser("add", help="add a note")
add.add_argument("text")
lst = sub.add_parser("list", help="list notes")
lst.add_argument("--tag")
rm = sub.add_parser("delete", help="delete a note by number")
rm.add_argument("id", type=int)

args = parser.parse_args(["delete", "3"])
assert (args.command, args.id) == ("delete", 3)
```

Each subcommand gets its own parser, with its own arguments and its own `--help`, so `notes delete --help` describes only `delete`.

`dest="command"` records which subcommand was chosen, and `required=True` makes running `notes` alone a usage error rather than doing nothing. A common pattern is to set `func=` on each subparser with `set_defaults` and call `args.func(args)`, so `main` needs no `if` chain over command names. The same layout scales to tools as big as `git`.

CHECKLIST

- ☐ Can you name a program's three inputs and three outputs, and what each is for?
- ☐ Can you build an argparse interface with positional arguments, typed options and good help?
- ☐ Can you make a tool read standard input when no files are given?
- ☐ Can you choose exit statuses and send errors to standard error?

- ☐ Can you structure a tool around `main(argv)` so it can be tested and installed as a command?

GO DEEPER

GUIDE [Command Line Interface Guidelines](#) Prasad, Firshman, Tashian & Parish

DOCS [Argparse Tutorial](#) Python documentation

GUIDE [Creating and packaging command-line tools](#) Python Packaging User Guide

Programs with state

3 exercises

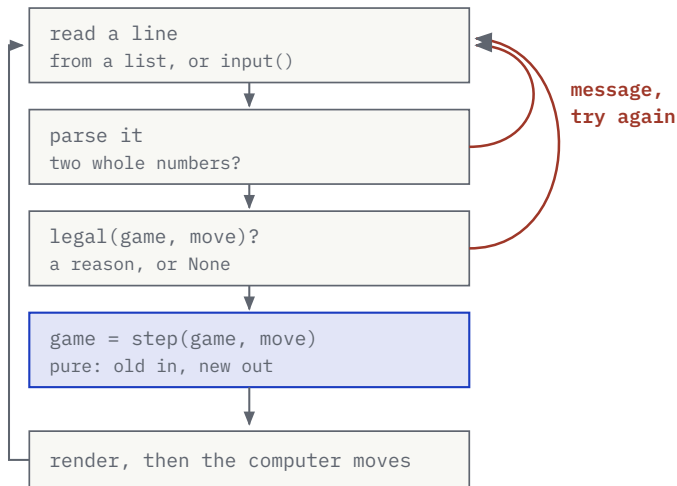
*A game, an editor, a shopping basket and a server all remember things between one input and the next. That memory is their **state**, and programs with state go wrong in characteristic ways: a change made in one place and forgotten in another, a move applied twice, a bug that only appears after a particular sequence of actions. The cure is a design: keep the whole state in one value, and make every change a pure function from the old state and an action to the new state. This chapter builds the game of Nim that way, and the design pays for itself at once in testing, replay, saving, undo and a computer opponent.*

State as a value

In Nim, players take turns removing any number of stones from one pile, and whoever takes the last stone wins. The example's whole state is a frozen dataclass, `Game`: the piles, whose turn it is, and the winner if there is one. Nothing about the game lives anywhere else, in globals or scattered across objects, so the question "what's the situation now?" always has a single, printable answer. Because the dataclass is frozen, a state can never change after it's made; each move makes a new one, which is what makes the rest of the chapter possible.

All the rules live in two functions. `legal(game, pile, take)` returns a reason a move isn't allowed, or `None`. `step(game, pile, take)` checks it, and returns the next state: the piles with stones removed, the turn passed to the other player, and the winner set if no stones remain. `step` is pure: the same state and move always give the same result, and nothing else changes. Everything else, reading what the player typed, printing the board, choosing the computer's move, is outside the rules, and talks to them only through those two functions.

THE LOOP



HISTORY: EVERY STATE KEPT

(3, 4, 5) you	[0]
(3, 2, 5) computer	[1]
(3, 2, 1) you	[2]
(0, 2, 1) computer	[3]
(0, 1, 1) you	[4] undo: [-3]
(0, 0, 1) computer	[5]
(0, 0, 0) computer won	[6]

saving is `json.dumps` of one state

The loop and its history. Input and output stay at the edges; the rules are one pure function; and because states never change, keeping every one of them is free, which makes undo and replay trivial.

EXAMPLE

state.py · Nim as an immutable state and a pure step function, a loop fed from a list, invalid input handled, save and load as JSON, undo from history, and an opponent that plays perfectly

```

import json, random
from dataclasses import dataclass, replace, asdict
from functools import reduce

# Nim: players take turns removing any number of stones from one pile; whoever takes the last
# wins.
@dataclass(frozen=True)
class Game:
    piles: tuple
    turn: str = "you" # whose move it is
    winner: str | None = None

def legal(game, pile, take):
    if game.winner: return "the game is over"
    if not 0 <= pile < len(game.piles): return f"there is no pile {pile + 1}"
    if take < 1: return "take at least 1"
    if take > game.piles[pile]: return f"pile {pile + 1} only has {game.piles[pile]}"
    return None

def step(game, pile, take):
    """The whole of the rules: a pure function from (state, move) to the next state."""
    problem = legal(game, pile, take)
    if problem: raise ValueError(problem)
    piles = game.piles[:pile] + (game.piles[pile] - take,) + game.piles[pile + 1:]
    other = "computer" if game.turn == "you" else "you"
    return replace(game, piles=piles, turn=other, winner=game.turn if sum(piles) == 0 else
        None)

def best_move(game, rng):
    """Leave the piles with an XOR of zero if possible: that position is a loss for the
    mover."""
    x = reduce(lambda a, b: a ^ b, game.piles)
  
```

```

for i, p in enumerate(game.piles):
    if p ^ x < p: return i, p - (p ^ x)
i = rng.choice([i for i, p in enumerate(game.piles) if p]) # losing anyway: play for
time
return i, 1

def show(game): return " ".join(f"{i + 1}:{' | ' * p:<5}" for i, p in
    enumerate(game.piles)).rstrip()

def play(game, inputs, rng, history):
    """The loop: read a move, check it, apply it. Input comes from a list, so it can be
    replayed."""
    for text in inputs:
        if game.winner: break
        try: pile, take = (int(part) for part in text.split())
        except ValueError:
            print(f"    you typed {text!r}: type a pile number and how many to take, like '2
                3'")
            continue
        try: game = step(game, pile - 1, take)
        except ValueError as e: print(f"    you typed {text!r}: {e}"); continue
        history.append(game)
        print(f"    {f'you take {take} from pile {pile}':<30} {show(game)}")
        if game.winner: break
        pile, take = best_move(game, rng)
        game = step(game, pile, take); history.append(game)
        print(f"    {f'computer takes {take} from pile {pile + 1}':<30} {show(game)}")
    return game

rng = random.Random(1)
start = Game(piles=(3, 4, 5))
history = [start]
print(f"start: {show(start)}")
game = play(start, ["2 2", "9 1", "1 x", "1 3", "3 7"], rng, history)

saved = json.dumps(asdict(game)) # state is plain data, so saving is one
line
print(f"\nsaved: {saved}")
loaded = Game(**{**json.loads(saved), "piles": tuple(json.loads(saved)["piles"])})
print(f"loaded == game: {loaded == game}")
game = play(loaded, ["3 2", "2 1", "3 1"], rng, history)
print(f"winner: {game.winner}")

previous = history[-3] # every state was kept, so undo is a
lookup
print(f"\nundo your last move and the computer's reply: {show(previous)}, {previous.turn} to
move")
try: step(game, 0, 1)
except ValueError as e: print(f"moving after the end: ValueError: {e}")

```

OUTPUT

```

start: 1:| | |    2:| | | |    3:| | | | |
    you take 2 from pile 2          1:| | |    2:| |    3:| | | | |
    computer takes 4 from pile 3    1:| | |    2:| |    3:|
    you typed '9 1': there is no pile 9
    you typed '1 x': type a pile number and how many to take, like '2 3'
    you take 3 from pile 1          1:          2:| |    3:|
    computer takes 1 from pile 2    1:          2:|      3:|
    you typed '3 7': pile 3 only has 1

saved: {"piles": [0, 1, 1], "turn": "you", "winner": null}
loaded == game: True
    you typed '3 2': pile 3 only has 1
    you take 1 from pile 2          1:          2:          3:|
    computer takes 1 from pile 3    1:          2:          3:

```

```
winner: computer
```

```
undo your last move and the computer's reply: 1:      2:|      3:|, you to move  
moving after the end: ValueError: the game is over
```

What the design buys

Bad input can't break anything. The loop parses each line, and if it isn't two numbers, or names a pile that doesn't exist, or takes more stones than a pile has, it prints a specific message and asks again; the state is untouched, because nothing changes until `step` succeeds. The example feeds it three kinds of bad input, four times, and the game carries on.

The game can be replayed and tested. `play` reads moves from a list, so the same list always produces the same game. A real game passes lines typed at the keyboard, `iter(input, "quit")` for instance, through the same function. Tests don't need a person: call `step` on a state and check the result, or replay a whole game from a list and check the winner. The computer's random choices use a seeded generator, so even its play is reproducible, as chapter 11 requires.

Saving is one line, because the state is plain data: `json.dumps(asdict(game))`. Loading needs one detail, turning the JSON list back into a tuple, and the loaded state compares equal to the original, so the game continues exactly where it left off. **Undo is a lookup**: the loop appends every state to `history`, which costs almost nothing because states are small and never change, and stepping back two entries undoes your move and the computer's reply.

The opponent is a function of the state too. Nim has a complete strategy, found by Charles Bouton in 1901: XOR all the pile sizes together, and if the result isn't zero, there is always a move that makes it zero, which leaves your opponent in a losing position. `best_move` finds it in four lines. Your first move, taking 2 from pile 2, left a non-zero XOR, so the computer won from there; exercise 19.1 finds the move that would have won.

This shape, one immutable state, actions and a pure update function, is how many interactive programs are built. The Elm language and the Redux library for web interfaces are organised around exactly it, and the same idea appears in databases as an event log that can be replayed to rebuild any past state. When a program with state becomes hard to reason about, moving toward this design is usually the fix.

COMMON TRAPS

State scattered across globals and objects, so no single place says what the situation is.

Rules mixed into input handling and display, so they can't be tested or reused.

Changing the state before validating the move, leaving it half-updated when validation fails.

Reading input straight from `input()` deep in the logic, so the program can't be replayed or tested.

Unseeded randomness, so a bug that appeared in one game can't be reproduced.

EXERCISES

EXERCISE 19.1 The winning first move

From piles (3, 4, 5), with you to move, find the move that wins against perfect play, by hand with XOR. Then check, with `step` and XOR, that the position it leaves has no winning reply.

SOLUTION

In binary the piles are 011, 100 and 101; XOR them and you get 010, which is 2. A winning move must leave an XOR of 0, so look for a pile p where $p \text{ XOR } 2$ is smaller than p : for 3, $3 \text{ XOR } 2 = 1$, which is smaller, so reduce pile 1 from 3 to 1 by taking 2. (For 4, $4 \text{ XOR } 2 = 6$ and for 5, $5 \text{ XOR } 2 = 7$, both larger, so neither works.) The result, (1, 4, 5), has XOR $1 \oplus 4 \oplus 5 = 0$. After `step(Game((3, 4, 5)), 0, 2)` the computer starts from a zero XOR, can only leave a non-zero one, and every time you restore zero with `best_move`'s rule you stay winning until you take the last stone. The rule works because any move from a zero-XOR position changes exactly one pile, which must make the XOR non-zero, while from a non-zero position some move always restores zero, and the empty position has XOR zero.

EXERCISE 19.2 Testing the rules

Write tests for `step`: one for a normal move, one for each illegal move, one that the game ends with the right winner, and a property check that in any random sequence of legal moves the total number of stones strictly decreases and never goes negative.

SOLUTION

```
import random, pytest

def test_a_move_takes_stones_and_passes_the_turn():
    assert step(Game((3, 4)), 1, 2) == Game((3, 2), turn="computer")

@pytest.mark.parametrize("pile, take", [(5, 1), (0, 0), (0, 4)])
def test_illegal_moves_are_refused(pile, take):
    with pytest.raises(ValueError):
        step(Game((3, 4)), pile, take)
```

```
def test_taking_the_last_stone_wins():
    assert step(Game((0, 2)), 1, 2).winner == "you"

def test_stones_always_decrease():
    rng = random.Random(3)
    for _ in range(200):
        game = Game(tuple(rng.randint(0, 6) for _ in range(3)))
        while sum(game.piles) and not game.winner:
            pile = rng.choice([i for i, p in enumerate(game.piles) if p])
            before, game = sum(game.piles), step(game, pile, rng.randint(1,
                game.piles[pile]))
            assert 0 <= sum(game.piles) < before
```

Every test builds a state directly and calls `step`, with no input, output or game loop involved, which is the payoff of keeping the rules pure. The property check plays 200 random games and checks after every move that stones were removed and none went negative, which also proves every game ends, since a non-negative whole number can't decrease forever: it's chapter 3's loop variant, applied to a whole game.

EXERCISE 19.3 Undo and redo

Add undo and redo commands to the game: `u` undoes your last move and the computer's reply, `r` redoes them, and making a new move after an undo discards the redo history. Which data structure do you need?

SOLUTION

Two stacks of states, as every editor does it. Keep `past`, the states before the current one, and `future`, the states undone. Undo pops the current state onto `future` and takes the top of `past` as current, twice to cover both players' moves. Redo does the reverse. Any new move pushes the current state onto `past`, applies `step`, and clears `future`, because the undone moves no longer follow from where you are. All of this is possible only because states are immutable values: undo just means "go back to that value", with no need to reverse each change, which is what makes undo so hard to add to programs that mutate their state in place. The loop handles `u` and `r` before trying to parse a move, and refuses with a message when the relevant stack is empty.

CHECKLIST

- ☐ Can you represent a program's whole state as one immutable value?

- ☐ Can you write state changes as a pure function from state and action to a new state?
- ☐ Can you validate input with helpful messages without ever leaving the state half-changed?
- ☐ Can you save and load state as JSON, and implement undo with a history of states?
- ☐ Can you test the rules directly, and replay a session from a list of inputs?

GO DEEPER

DOCS [Redux: Three Principles, one state, read-only, pure updates](#) The Redux team

BOOK [Game Programming Patterns: State](#) Robert Nystrom

ARTICLE [Nim, and Bouton's winning strategy](#) Wikipedia

Talking to APIs

3 exercises

Calling a web API is one line of code, and making that line dependable takes the rest of the program. The network is slow sometimes and absent sometimes; servers fail briefly, refuse requests they consider too frequent, and return errors for things that don't exist. And asking again for data you already have wastes everyone's time. This chapter builds a client that sets timeouts, retries only what's worth retrying, honours rate limits, and caches answers on disk, revalidating them cheaply when they get old. It runs against a small API server started inside the example, so every run behaves the same without touching the internet.

Requests and responses

A web API speaks HTTP. The client sends a **request**: a method, usually `GET` to read and `POST` to create, a URL, and headers. The server sends a **response**: a status code, headers, and a body, which for most APIs is JSON. The status code's first digit is its class. **2xx** means success: 200 OK. **3xx** means look elsewhere, including 304 Not Modified, "your copy is still current". **4xx** means the client's request was wrong: 404 Not Found, 401 for missing credentials, and 429 Too Many Requests. **5xx** means the server failed: 500, or 503 Service Unavailable. That classification tells the client what to do next, and getting it right is most of what this chapter is about.

Python's standard library includes `urllib.request`, which the example uses so it runs without installing anything. Most Python projects use the third-party `requests` or `httpx` libraries instead, which have friendlier interfaces; everything below applies to them in the same way. One quirk of `urllib` is that it raises `HTTPError` for any status that isn't 2xx, even 304, which the example's client handles explicitly.

EXAMPLE

`api.py` · a JSON API served locally, and a client with timeouts, retries with backoff, Retry-After for rate limits, a disk cache, ETag revalidation, and errors raised for what can't be fixed by trying again

```
import hashlib, json, pathlib, tempfile, threading, time, urllib.error, urllib.request
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

# A small API running on this machine, so the example behaves the same everywhere
DATA = {"/cities/oslo": {"city": "Oslo", "temp_c": 4}, "/cities/lima": {"city": "Lima",
    "temp_c": 19},
    "/slow": {"slow": "but finished"}}
log, flaky = [], {"/cities/lima": 2} # lima fails twice with 503, then works
slow_done = threading.Event()
class API(BaseHTTPRequestHandler):
    def do_GET(self):
```

```

if self.path == "/slow": time.sleep(0.5) # longer than the client will wait
if self.path == "/busy" and sum(p == "/busy" for _, p, _ in log) == 0:
    return self.reply(429, {"error": "slow down"}, {"Retry-After": "2"})
if flaky.get(self.path, 0) > 0:
    flaky[self.path] -= 1
    return self.reply(503, {"error": "try again"})
body = DATA.get(self.path, {"busy": "ok"} if self.path == "/busy" else None)
if body is None: return self.reply(404, {"error": f"no such city: {self.path}"})
tag = "" + hashlib.sha256(json.dumps(body).encode()).hexdigest()[:12] + ""
if self.headers.get("If-None-Match") == tag: return self.reply(304, None, {"ETag":
    tag})
self.reply(200, body, {"ETag": tag})
def reply(self, status, body, headers=None):
    log.append((self.command, self.path, status))
    data = json.dumps(body).encode() if body is not None else b""
    try:
        self.send_response(status)
        for k, v in (headers or {}).items(): self.send_header(k, v)
        self.send_header("Content-Type", "application/json")
        self.send_header("Content-Length", str(len(data)))
        self.end_headers(); self.wfile.write(data)
    except (BrokenPipeError, ConnectionResetError): pass # the client gave up waiting
    if self.path == "/slow": slow_done.set()

def log_message(self, *args): pass # keep the server quiet
server = ThreadingHTTPServer(("127.0.0.1", 0), API) # port 0: let the system pick a free
one
threading.Thread(target=server.serve_forever, daemon=True).start()
BASE = f"http://127.0.0.1:{server.server_address[1]}"

class Client:
    """Fetches JSON with a timeout, retries with backoff, honours Retry-After, and caches on
    disk."""
    def __init__(self, cache_file, clock, sleep, max_age=60):
        self.cache_file, self.clock, self.sleep, self.max_age = cache_file, clock, sleep,
        max_age
        self.cache = json.loads(cache_file.read_text()) if cache_file.exists() else {}
    def get(self, path, timeout=2.0):
        entry = self.cache.get(path)
        if entry and self.clock() - entry["at"] < self.max_age:
            return entry["body"], "cache, fresh"
        headers = {"If-None-Match": entry["etag"]} if entry else {}
        for attempt in range(4):
            try:
                req = urllib.request.Request(BASE + path, headers=headers)
                with urllib.request.urlopen(req, timeout=timeout) as r:
                    body, etag, how = json.load(r), r.headers["ETag"], "network"
                break
            except urllib.error.HTTPError as e:
                if e.code == 304: # not modified: reuse ours
                    body, etag, how = entry["body"], entry["etag"], "cache, revalidated"; break
                if e.code == 429: self.sleep(float(e.headers["Retry-After"])); continue
                if e.code >= 500 and attempt < 3: self.sleep(0.5 * 2 ** attempt); continue
                raise RuntimeError(f"{path}: HTTP {e.code} {json.load(e).get('error')}") from e
        self.cache[path] = {"at": self.clock(), "etag": etag, "body": body}
        self.cache_file.write_text(json.dumps(self.cache))
        return body, how

now, slept = [1000.0], [] # a fake clock and a record of
waits
with tempfile.TemporaryDirectory() as folder:
    client = Client(pathlib.Path(folder, "cache.json"), clock=lambda: now[0],
        sleep=slept.append)
    for path, advance in ((("/cities/oslo", 0), ("/cities/oslo", 30), ("/cities/oslo", 61),
        ("/cities/lima", 0), ("/busy", 0))):
        now[0] += advance
        body, how = client.get(path)

```

```

    print(f"t={now[0] - 1000:>3.0f}s GET {path:<14} {how:<20} {body}")
    print(f"waits requested by the client: {slept}")
    again = Client(pathlib.Path(folder, "cache.json"), clock=lambda: now[0],
        sleep=slept.append)
    print(f"a new client reads the saved cache: {again.get('/cities/lima')}")
    for path, timeout in (("/cities/rome", 2.0), ("/slow", 0.2)):
        try: client.get(path, timeout=timeout)
        except RuntimeError as e: print(f"error: {e}")
        except TimeoutError: print(f"error: {path}: no answer within {timeout} s")
    slow_done.wait(5) # let the slow request finish
    anyway
    print("\nwhat the server saw:")
    for method, path, status in log: print(f"    {method} {path:<14} {status}")
    server.shutdown()

```

OUTPUT

```

t= 0s GET /cities/oslo    network    {'city': 'Oslo', 'temp_c': 4}
t= 30s GET /cities/oslo    cache, fresh    {'city': 'Oslo', 'temp_c': 4}
t= 91s GET /cities/oslo    cache, revalidated    {'city': 'Oslo', 'temp_c': 4}
t= 91s GET /cities/lima    network    {'city': 'Lima', 'temp_c': 19}
t= 91s GET /busy          network    {'busy': 'ok'}
waits requested by the client: [0.5, 1.0, 2.0]
a new client reads the saved cache: ({'city': 'Lima', 'temp_c': 19}, 'cache, fresh')
error: /cities/rome: HTTP 404 no such city: /cities/rome
error: /slow: no answer within 0.2 s

what the server saw:
    GET /cities/oslo    200
    GET /cities/oslo    304
    GET /cities/lima    503
    GET /cities/lima    503
    GET /cities/lima    200
    GET /busy          429
    GET /busy          200
    GET /cities/rome    404
    GET /slow          200

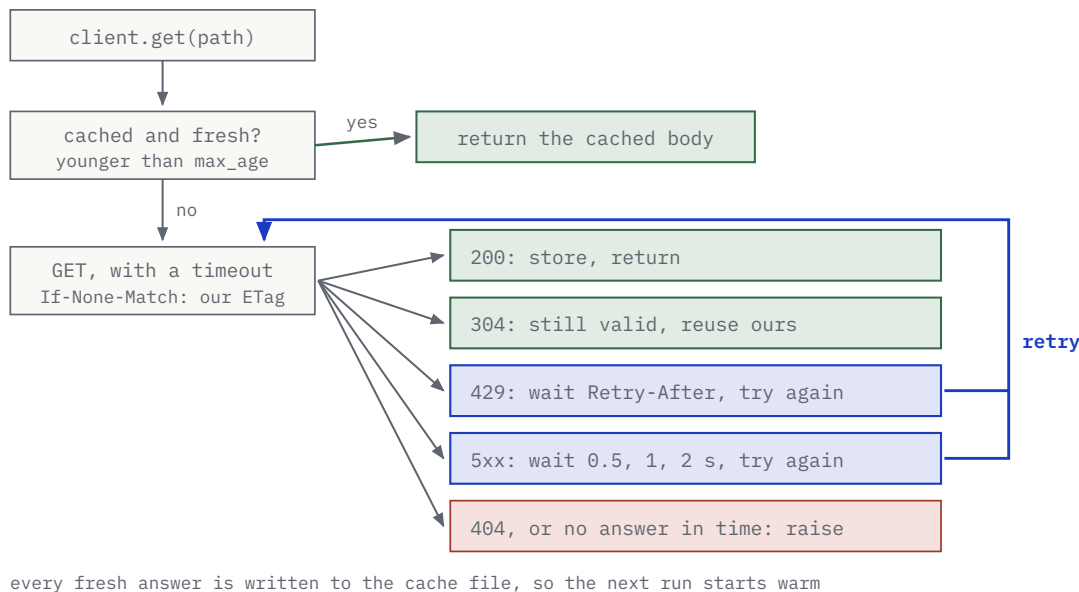
```

Failing well: timeouts, retries and rate limits

Always set a timeout. Without one, a request to a server that has stopped responding waits for minutes, or forever, and your program hangs with it. The example's `/slow` endpoint takes half a second and the client allows 0.2, so it gives up and reports "no answer within 0.2 s". Note in the server's log that the slow request still finished, with a 200 nobody received: a timeout means you stopped waiting, not that the server stopped working, which matters for requests that change things.

Retry only what might succeed next time. A 503 or 500 is often momentary: a server restarting, a database briefly overloaded. The client retries those up to three times, waiting 0.5, then 1, then 2 seconds, which is **exponential backoff**: each wait doubles, so a struggling server gets more breathing room, not a flood of retries. The example's `/cities/lima` fails twice and succeeds on the third attempt, and the waits recorded are 0.5 and 1.0. A 404 is different: asking again won't make Rome exist in the server's data, so the client raises an error at once. Retrying 4xx errors only adds load and delay. Retries are also only safe for requests that can be repeated without harm, like `GET`; a retried "charge the card" request can charge twice, which is why payment APIs use idempotency keys.

Honour rate limits. A server returning 429 is saying you've asked too often, and usually includes a `Retry-After` header saying how long to wait, here 2 seconds. The client waits exactly that and tries again. Ignoring it typically gets the client blocked. The example's client takes its `sleep` function as a parameter, so the run records the waits instead of actually sleeping, the same injection of dependencies as chapter 11, which is why the example runs in about a second.



What the client does with each outcome. Fresh cache hits never touch the network; stale ones are revalidated cheaply; transient failures are retried with growing waits; permanent ones are reported at once.

Caching

The second request for Oslo, 30 seconds after the first, never reached the server: the cached answer was younger than the client's `max_age` of 60 seconds, so it was used as it was. That's **freshness**, and it's the biggest saving a cache offers. Choosing `max_age` is a judgement about the data: a city's weather can be a few minutes old, an exchange rate perhaps seconds, a country's name for ever.

After 91 seconds the cached copy was stale, but probably still correct, and downloading it again would waste bandwidth. So the client sends a **conditional request**: `If-None-Match` with the **ETag**, a fingerprint of the content that the server sent with the original response. If the content hasn't changed, the server replies 304 with no body, and the client reuses its copy and resets its age; that's **revalidation**. The server log shows it: the second request for Oslo got 304. The cache lives in a JSON file, so a new client, or the next run of the program, starts with everything the last one learned, as the line "a new client reads the saved cache" shows.

Two practical rules complete the picture. Keep credentials such as API keys out of your code and out of the cache: read them from an environment variable, `os.environ["WEATHER_API_KEY"]`, and never print or log them. And in unit tests, never call

the real API, which makes tests slow, flaky and dependent on someone else's server; run a local server as the example does, or pass the client a fake function in place of the network.

COMMON TRAPS

No timeout, so one unresponsive server hangs the whole program.

Retrying every error, immediately and forever, including 4xx errors that will never succeed.

Retrying requests that aren't safe to repeat, such as payments, without an idempotency key.

Ignoring `Retry-After` and getting rate-limited further, or blocked.

API keys in source code, where they end up in version control and in logs.

EXERCISES

EXERCISE 20.1 Jitter

A thousand clients all hit a server's outage at the same moment and use the example's backoff, 0.5, 1 and 2 seconds. What happens when the server comes back, and how does adding random **jitter** to the waits help? Write the changed line.

SOLUTION

```
self.sleep(random.uniform(0, 0.5 * 2 ** attempt))    # "full jitter"
```

With fixed waits, all thousand clients retry at exactly 0.5 seconds, then all at 1.5, then all at 3.5: synchronised waves of a thousand requests each, which can knock a recovering server straight back over. That's called a **thundering herd**. Choosing each wait at random between 0 and the backoff limit spreads the retries evenly across the interval, so the server sees a steady trickle instead of spikes; AWS's architecture blog popularised this "full jitter" variant after measuring it against the alternatives. In tests, the random generator should be passed in and seeded, like the clock and sleep, so the waits are still reproducible.

EXERCISE 20.2 How long to cache

Choose a `max_age` for each, and say whether revalidation with ETags is worth it: today's weather forecast, a currency exchange rate used to show prices, the list of countries, and a user's own profile that they just edited.

SOLUTION

A weather forecast changes a few times an hour, so 10 to 30 minutes is reasonable, and revalidation helps because unchanged forecasts are common. An exchange rate used for display tolerates seconds to a minute of staleness, but if it's used to charge someone, don't cache it at all, or fix the rate explicitly for that transaction. The list of countries changes almost never: cache it for a day or a week and revalidate, since it's large and nearly always unchanged. A user's own profile right after they edited it must not be served from a cache, or they'll see their change "fail": either bypass the cache for their own data, or update the cached copy with what they just saved, which is called write-through. There's no universal right answer; the question is always how wrong the data may be and for how long, and what that costs.

EXERCISE 20.3 Testing without the network

Restructure the client so that tests can replace the HTTP call itself, without running any server. Show a test that checks a 503 is retried and a 404 isn't.

SOLUTION

```
class Client:
    """The retry logic, with the HTTP call passed in. (Caching works as before, and is
    left out.)"""
    def __init__(self, fetch, sleep=time.sleep):    # fetch(path, headers) ->
        (status, headers, body)
        self.fetch, self.sleep = fetch, sleep
    def get(self, path):
        for attempt in range(4):
            status, headers, body = self.fetch(path, {})
            if status == 200: return body, headers.get("ETag")
            if status == 429: self.sleep(float(headers["Retry-After"])); continue
            if status >= 500 and attempt < 3: self.sleep(0.5 * 2 ** attempt); continue
            raise RuntimeError(f"{path}: HTTP {status}")

    def fake(responses):
        calls = []
        def fetch(path, headers):
            calls.append(path)
            return responses.pop(0)
        return fetch, calls

    def test_retries_503_but_not_404():
        fetch, calls = fake([(503, {}, None), (200, {"ETag": '"x"', "ok": 1})])
        assert Client(fetch, sleep=lambda s: None).get("/a")[0] == {"ok": 1} and calls ==
            ["/a", "/a"]
```

```
fetch, calls = fake([(404, {}, {"error": "no"})])
with pytest.raises(RuntimeError):
    Client(fetch, sleep=lambda s: None).get("/b")
assert calls == ["/b"]
```

Separating "make one HTTP request" from "decide what to do with the response" puts all the interesting logic, retries, backoff, caching, in code that a test can drive with a list of canned responses, in microseconds. The real `fetch` is a few lines wrapping `urllib` or `httpx`, tested once against a local server like the example's. The test checks behaviour by counting calls: one retry after the 503, none after the 404. This is the same design as chapter 19's pure `step` function, applied to I/O: keep decisions pure, push the side effects to the edges, and pass them in.

CHECKLIST

- ☐ Can you read an HTTP status code by its class and decide what a client should do?
- ☐ Can you set timeouts, and retry transient failures with exponential backoff and jitter?
- ☐ Can you honour `Retry-After` and explain which requests are safe to retry?
- ☐ Can you cache responses with a freshness limit and revalidate them with ETags?
- ☐ Can you test network code without the network, and keep credentials out of code?

GO DEEPER

DOCS [MDN: HTTP caching](#) Mozilla

ARTICLE [Exponential Backoff And Jitter](#) Marc Brooker, AWS

DOCS [HTTPX: a next-generation HTTP client for Python](#) Encode

The stage asks for four programs, each a different shape, each with tests, a README and no framework doing the thinking for you. Chapters 18, 9, 19 and 20 build a small working version of each; the lists below say what the full versions need. Write them in that order or any other, but finish each one: a program that someone else has installed, run and used teaches more than four that nearly work.

1. A command-line tool

Pick something you'd use yourself: a to-do list, a unit converter, a tool that renames photos by date, a log summariser. Chapter 18 is the model.

- 01 Arguments parsed with `argparse`, a `--help` screen with a sentence for every option, and `--version`.
- 02 Reads standard input when no file is given, so it works in a pipeline.
- 03 Results on standard output, errors on standard error, and exit statuses 0, 1 and 2 used as chapter 18 describes.
- 04 All logic behind `main(argv)`, with tests that call it directly and at least one that runs it as a process.
- 05 Installable as a command through `[project.scripts]` in `pyproject.toml`.

2. Messy data to a clean report

Download a real public dataset in CSV, from a city's open-data portal or a national statistics office, and produce a report from it. Chapter 9 is the model.

- 01 A cleaning function that returns a clean record or raises `ValueError` with a reason, and unit tests for each kind of mess the file contains.
- 02 Rejected rows written to an errors file with their line numbers and reasons, and a check that clean rows plus rejected rows equals all rows.
- 03 Explicit decisions, in comments and tests, about every ambiguity: date formats, units, duplicates, missing values.

04 Streaming: memory use that doesn't grow with the number of rows.

05 The report as CSV or JSON for programs, and as readable text for people.

3. A small game with state

Nim, noughts and crosses, hangman, Mastermind, a tiny text adventure. Chapter 19 is the model.

01 The whole state in one immutable value, and every change through a pure `step(state, action)`.

02 Input validated with a specific message for each mistake, and no input that can crash the program.

03 A game loop that reads from any iterable, so a test can replay a whole game from a list.

04 Save and load as JSON, and undo.

05 Randomness from a seeded generator passed in, and tests for every rule, including a property check over random games.

4. A script that talks to a public API

Choose a free API, such as Open-Meteo for weather, which needs no key, or the GitHub REST API, and build something useful on it: a forecast for your city, a summary of a repository's recent activity. Chapter 20 is the model.

01 A timeout on every request, retries with exponential backoff and jitter for 5xx errors and timeouts only, and `Retry-After` honoured for 429.

02 A disk cache with a `max_age` chosen and justified for the data, and ETag revalidation where the API supports it.

03 Any API key read from an environment variable, never stored in the code, the cache or the logs.

04 Tests that never touch the network, using a fake fetch function or a local

04 Tests that never touch the network, using a fake fetch function of a local server.

05 Clear messages for the failures a user will actually meet: no network, a bad key, a rate limit.

For every program: a README that says what it does, how to install it, and how to run it, with an example; tests that run with one command; and a virtual environment with a lockfile. Then give each to someone else, watch them use it without your help, and fix whatever confused them.

THE TWENTY RULES

One per chapter, each linked back to it. Read them before you start a new program, and again when one surprises you.

-
- 01 Predict what code will do before you run it, and know the type of every value.
-
- 02 A name points to an object: know whether you're rebinding the name or mutating the object.
-
- 03 Know each loop's invariant and why it stops, and use half-open ranges.
-
- 04 One function, one job: return results, and keep printing and other side effects at the edges.
-
- 05 Every recursion needs a base case and a shrinking step; memoise repeated work, and use a loop when it runs deep.
-
- 06 Indexing and appending are cheap; inserting at the front and `in` are not. Choose the structure by its operations.
-
- 07 Look things up in a dict or set, and make equal keys hash equally.
-
- 08 Decode at the edge, work in `str`, encode at the edge, and always name the encoding.
-
- 09 Open files with `with` and an encoding; reject bad rows with reasons, and never guess.
-
- 10 Catch narrowly, validate at the boundary, and put the bad value in the message.
-
- 11 Test from week one: edges first, seeded randomness, and watch every new test fail.
-
- 12 Look, don't guess: reproduce, stop the program, inspect; bisect when you don't know where to look.
-
- 13 A class owns its data and its invariants; compute derived values with

properties.

-
- 14 Prefer composition and narrow interfaces; inherit only from classes designed for it.
 - 15 Copy mutable data at boundaries, and never build nested lists with `*`.
 - 16 One virtual environment per project; ranges for libraries, lockfiles for applications.
 - 17 Map code before you read it, and record what it does before you change it.
 - 18 Results to standard output, messages to standard error, and an honest exit status.
 - 19 Keep state in one immutable value, and change it with a pure function.
 - 20 Set a timeout, retry only what might succeed, and cache what you've already been told.
-

READING LIST

The books, courses and documentation behind this volume, and the chapters each one deepens. If you do only one thing, do Berkeley's CS 61A with its free textbook, *Composing Programs*, all the way through, with the exercises. For daily reference, the official Python tutorial and documentation are better than most books.

WORK	AUTHOR	DEEPENS
Composing Programs	John DeNero	Ch. 1–5, 13
CS 61A: Structure and Interpretation of Computer Programs	UC Berkeley	Ch. 1–5, 13–14
Structure and Interpretation of Computer Programs	Abelson & Sussman	Ch. 4–5, 19
CS50x: Introduction to Computer Science	Harvard	Ch. 3, 12, 15
The Python Tutorial	Python documentation	Ch. 1–10, 13, 16
Exercism: the Python track	Exercism	practice for every chapter
Automate the Boring Stuff with Python	Al Sweigart	Ch. 9, 18, 20
Fluent Python	Luciano Ramalho	Ch. 6–8, 13–15
Effective Python	Brett Slatkin	Ch. 4, 10, 13–14
Python Testing with pytest	Brian Okken	Ch. 11
The Odin Project	The Odin Project community	if the web is your destination

Compiled October 2026 as the companion to stage 01, Programming foundations, of The Engineer's Syllabus, alongside *The Engineer's Workbench*, *Compass*, *Mill*, *Loom*, *Exchange*, *Press*, *Ledger*, *Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel* and *Foundry*. It follows the order of topics in the stage and draws on the works in the reading list, and on the original sources for its named results: Bouton (1901) on the strategy for Nim; Dijkstra (1982) on numbering from zero; Bentley (1986) and Bloch (2006) on binary search; Liskov (1987) on substitution; Fowler, Noll and Vo (1991) on the FNV hash; Thompson and Pike (1992) on UTF-8; Gamma, Helm, Johnson and Vlissides (1994) on composition over inheritance; Feathers (2004) on characterisation tests; and Brooker (2015) on backoff with jitter. Library versions and tools are described as of late 2026; Python changes slowly and carefully, and everything here should hold for years.

Every example was executed before publication with Python 3.14 on macOS, and they're written for Python 3.12 or later and to print the same output on any machine: random choices use fixed seeds, files live in temporary folders, the network example runs its own local server, and where a chapter is about cost it counts operations rather than timing them. Error messages are shown as Python 3.14 prints them; earlier versions word a few of them differently. Re-run in October 2026 with Python 3.14.7, every example printed exactly the output shown. Tested here: an expression evaluated step by step; binding, mutation and scope; insertion sort with its invariant, and three binary searches against every small case; a problem decomposed into pure functions; the call stack, memoisation and an explicit stack; the cost of growing and searching lists; a hash table with a weak and a strong hash; UTF-8 encoded by hand; a messy CSV cleaned with every rejection explained; exceptions, chaining and validation at the boundary; unit tests and a property check that found what they missed; scripted debugger sessions and bisection; a booking calendar that guards its invariants; inheritance against composition; aliasing, copies and garbage collection; a package, a circular import, a virtual environment and dependency resolution; a legacy module mapped and refactored under characterisation tests; a command-line tool run as a real program; Nim with save, replay and undo; and an API client with retries, rate limits and a revalidating cache.