Programming Fundamentals

Curated undergraduate study track (not accredited, no external exam). Original explanations and practice only, following the reviewed course handout. Python 3; every program shown is complete and runnable.

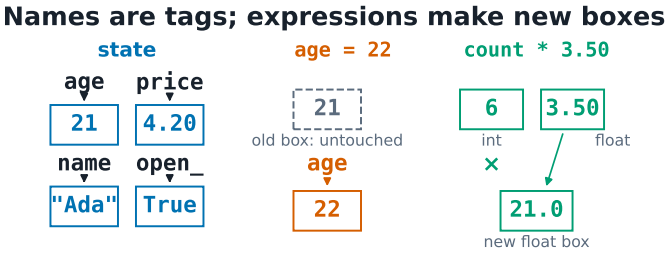
Predict, then run

- Every program in the course is complete and runnable — **type** it, run it, change one line.
- The predict-then-run loop is the entire skill.

Values, types and input/output: the type decides

1. **Types** "4" + "2" is "42"; 4 + 2 is 6; "4" + 2 is a TypeError — Python never guesses.
2. **Convert** count = int(input("Coffees? ")) — then count * 3.50 at 6 prints 21.0 (the .0 marks a float).
3. **Format** f"{name} has £{balance:.2f}" — formatting at display, the stored **value** unchanged.

State is name tags on value boxes

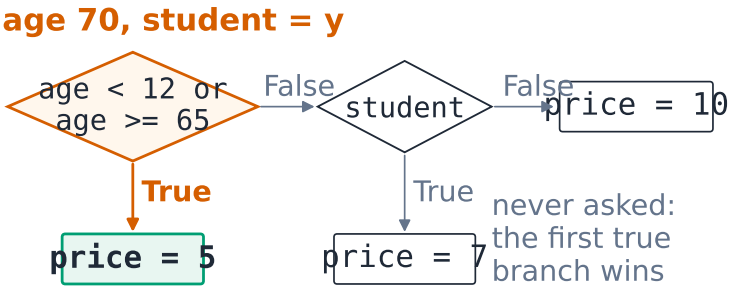


Names are tags; expressions read boxes and make new boxes; rebinding moves a tag, never edits a box.

Selection: the first true branch wins

1. **Price** age $<$ 12 or age $>=$ 65 $\rightarrow$ 5; elif student $\rightarrow$ 7; else $\rightarrow$ 10.
2. **Order** The age branch first, so a 70-year-old student pays 5 — the ordering IS the policy.
3. **Trap** $==$ compares, $=$ assigns; and 0.1 + 0.2 $==$ 0.3 is False — compare with abs(x $-$ y) $<$ 1e-9.

The first true branch decides the price



Age 70, student yes: the age branch matches first and the student branch is never reached.

Iteration: the trace table is the test

1. **Sum** for i in range(1, 101): total $+=$ i $\rightarrow$ 5050.
2. **Trace** The factorial while loop: (5,1) $\rightarrow$ (4,5) $\rightarrow$ (3,20) $\rightarrow$ (2,60) $\rightarrow$ (1,120) $\rightarrow$ prints 120.
3. **Check** Build the table by hand, then print inside the loop and let Python grade you.

Trace tables grade your mental model

```
n = 5
factorial = 1
while n > 1:
    factorial *= n
    n -= 1
print(factorial)
```

after	n	factorial	n > 1 ?
start	5	1	True
round 1	4	5	True
round 2	3	20	True
round 3	2	60	True
round 4	1	**120**	False

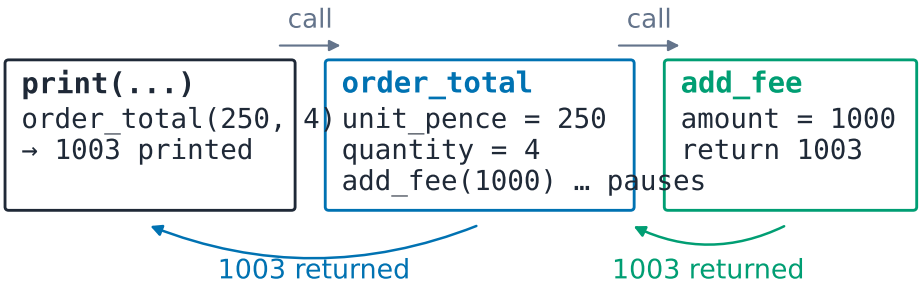
False: the loop ends and 120 is printed

> Round by round, n and factorial move together until the condition flips: four passes and the loop is done at 120.

Functions: return feeds the next function

1. **Define** def add_fee(amount): return amount + 3 — the docstring is the contract (whole pence in, out).
2. **Compose** def order_total(unit, qty): return add_fee(unit * qty) — order_total(250, 4) → 1003.
3. **Scope** Each call gets its own local names; they vanish when the call ends.

Nested calls pause and hand back



print calls order_total(250, 4), which pauses while add_fee(1000) runs; 1003 returns along the chain.

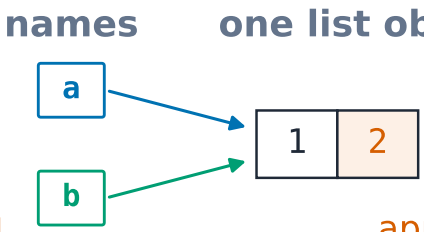
Collections: four shapes, four jobs

1. **Tally** tally[item] = tally.get(item, 0) + 1 — the **dictionary** keyed by the counted thing.
2. **Alias** a = [1,2]; b = a; b.append(3) → a is [1,2,3]. Copy with a[:] when independence matters.
3. **Safe loop** Mutating a **list** under **iteration** skips items — iterate a snapshot, marks[:].

Two names, one list

```
a = [1]
b = a
b.append(2)
```

a → [1, 2]



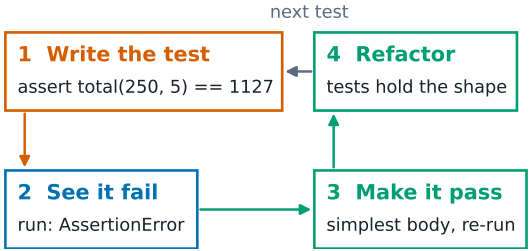
appended through b, seen through a

b = a shares the **list**: b.append(2) and a reads [1, 2] - assignment shares, never copies.

Files and errors: catch what you expect, by name

1. **Fallback** try: size = int(open("settings.txt").read()); except FileNotFoundError: size = 12.
2. **Convert** A **file** holding "big" makes int() raise ValueError — catch it too: size = 12.
3. **Let crash** The unexpected exception should stop the program loudly, not be swallowed.

Red, green, refactor



> Write the failing test; see it fail for the right reason; make it pass; refactor with the tests holding the shape.

Decomposition, testing and debugging: the discipline on top

1. **Test** assert order_total(250, 0) == 3 — the zero-quantity boundary; silence is a pass.
2. **Bisect** Reproduce small, inspect at the trust/suspect boundary, fix one thing, re-run.
3. **Wiring** Tests green, program wrong? Look between functions — swapped arguments, unstripped input.

Check yourself

- Types: 7 // 2 and 7 % 2? (3 and 1 — both int.)
- **Iteration**: 100 halved by // until 1 — how many loops? (Six.)
- **Testing**: order_total(250, 0) with a 3-fee? (3 — the fee still applies.)