

Programming Fundamentals

University Computer Science

What this course is, and how to use it

This is the first course in University Computer Science: **programming fundamentals**, taught with Python 3. It assumes no prior programming. Every program in this handout is complete and runnable – type it, run it, change one line, and predict what changes before running again. That predict-then-run loop is the entire skill.

The course is a ladder:

- Units 1.1 to 1.3 build the mechanics: values and types, selection, iteration.
- Unit 1.4 packages work into functions.
- Units 1.5 and 1.6 add the standard collections and files with error handling.
- Unit 1.7 is the discipline on top: decomposing problems, testing, debugging.

How to read each unit: a short story says why it matters; a picture or a rule of thumb comes before any syntax; then the precise rules; then worked examples that get harder, each with a trace you can check by running the code; then the mistakes people actually make; then a three-line summary. Every example here can be checked a second way – running it. Reading code without running it is like reading sheet music without hearing it.

Values, types and input/output

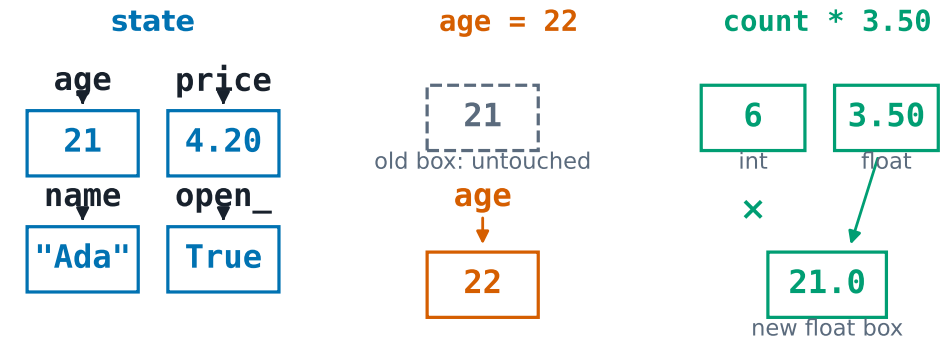
A shopping app shows “4.20” – is that text or a number? If it is text, adding two prices gives “4.203.50”; if it is a number, it gives 8.7. Which one happens is decided by the **type**, and the whole of unit 1.1 is learning to see types before you run anything.

A **value** has a type, and the type decides what you can do with it: **int** (whole numbers), **float** (decimals), **str** (text), **bool** (True/False). The **type(...)** function names any value’s type – ask it freely while learning.

```
age = 21           # int
price = 4.20       # float
name = "Ada"       # str
open_ = True       # bool
print(type(age))   # <class 'int'>
```

Variables are names bound to values; assignment `age = 22` rebinds the name – it never edits the old value. The mental picture, worth keeping for the whole degree:

Names are tags; expressions make new boxes



Names are tags on boxes; expressions read boxes and make new boxes

Reading an expression is “open the boxes, compute, close a new box”: `count * 3.50` with `count = 6` opens the **int** box 6 and the **float** box 3.50, and Python’s mixing rule promotes the **int** – the result is the **float** 21.0, never the **int** 21.

Conversion is explicit: `int("21")` is 21, but `"4" + 2` is a **TypeError** – Python refuses to guess whether you meant "42" or 6. Output with `print(...)`; input with `input(...)`, which always returns a **string**, so numbers need converting:

```
born = int(input("Birth year? "))
print("About", 2026 - born, "years old")
```

Worked example 1 (the café till). `count = int(input("Coffees? "))`, then `print("Total:", count * 3.50)`. At 6 coffees this prints `Total: 21.0` – the .0 marks a float, because `int * float` promotes to float. Watching which type comes out of an expression is unit 1.1’s whole habit. Trace it the state way: `input` returns the string "6"; `int(...)` opens a new box holding the **int** 6; the multiplication makes a fresh float box 21.0; `print` shows it.

Worked example 2 (the classic accident, repaired). A banner printer joins a count onto a message:

```
count = int(input("Tickets? "))
print("Sold " + count + " today")
```

Run it and Python raises **TypeError: can only concatenate str (not "int") to str** – the rule refuses to guess. The repair names the conversion explicitly: `print("Sold " + str(count) + " today")`, or the cleaner `print("Sold", count, "today")`, which inserts the spaces for you. The lesson to carry: `+` means addition for

numbers, joining for strings, and never a silent mix.

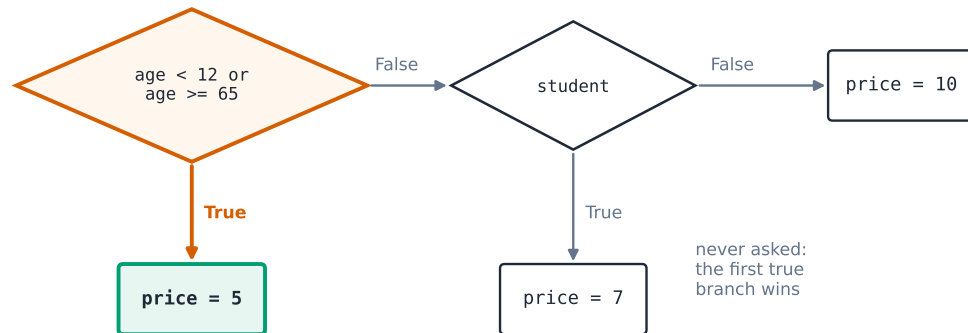
Common mistakes. Forgetting that `input()` always gives a string, so `input() * 2` repeats text rather than doubling a number. Writing `4.20 + 6` and expecting an `int` answer – any float in the expression makes a float out. Trusting `0.1 + 0.2 == 0.3` (it is `False`; binary rounding – unit 1.2 shows the fix).

Summary: four basic types decide every operation; names are tags, assignment rebinds; expressions mix types by promotion with floats; conversions are explicit and `input()` needs one.

Selection: choosing between paths

A theme park turnstile must charge children, adults and seniors differently – one program, several paths. **Selection** is how a program chooses: `if`, `elif`, `else` run a block only when a condition holds.

age 70, student = y



Age 70, student: the first true branch decides the price

The condition is any boolean expression; comparisons (`==`, `!=`, `<`, `>=`) and logic (`and`, `or`, `not`) build it. Two rules make every decision readable:

1. Branches are tested top to bottom; the **first true branch wins** and the rest are skipped.
2. Indentation is not decoration – it IS the block. One extra space changes which lines belong to the branch.

```
temp = 37.5
if temp >= 38:
    print("Fever")
elif temp >= 36 and temp < 38:
    print("Normal")
else:
    print("Low")
```

Worked example 1 (the cinema price). Under 12 or 65+ pay 5, students pay 7, everyone else 10:

```
age = int(input("Age? "))
student = input("Student? y/n ") == "y"
if age < 12 or age >= 65:
    price = 5
elif student:
    price = 7
else:
    price = 10
print("Pay", price)
```

Order matters: the first true branch wins, which is why the age check precedes the student check – a 70-year-old student should pay 5, and this ordering says so. Run it with 70 and y to see the flowchart's leftmost path taken.

Worked example 2 (grade bands, escalated). The same skeleton, now with a chain where each `elif` only needs one comparison, because the tests above it have already failed:

```
score = int(input("Score? "))
if score >= 80:
    band = "A"
elif score >= 70:
    band = "B"
elif score >= 60:
    band = "C"
else:
    band = "F"
print("Band:", band)
```

At 79 the first test fails, the second succeeds: **Band: B**. The escalation to notice: the `elif score >= 70` does not also need `and score < 80` – reaching that line already proves the score is below 80. Trusting the fall-through like this is what makes chains short; re-testing everything is the beginner's longer, wronger version.

Floating-point comparisons need care: `0.1 + 0.2 == 0.3` is `False` (binary rounding), so compare against a tolerance: `abs(x - y) < 1e-9`.

Common mistakes. Writing `=` where `==` is meant – the classic bug, and Python refuses it in conditions (`SyntaxError`) precisely because it is so common. Testing the special case after the general one, so the general one always wins. Using `elif` with a condition that can never be reached because an earlier branch already caught it.

Summary: conditions choose paths, first-true-wins, indentation is the block; order tests specific-to-general; floats compare by tolerance, never by `==`.

Iteration: doing it again

Paying 5,000 invoices by hand is a career; with a loop it is a program. **Iteration** repeats work: a `for` loop walks a known sequence; a `while` loop runs while a condition holds. The design question is always: do you know how many trips, or does a condition decide?

```
total = 0
for i in range(1, 101):      # 1..100 - range stops BEFORE its end
    total += i
print(total)                 # 5050
```

Use `for` when the trip count is known; use `while` when a condition decides. Every `while` needs three things visible in the code: the variable’s start, the condition, and something inside that moves toward exiting – otherwise it is an **infinite loop**.

```
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

Trace table for the factorial loop

```
n = 5
factorial = 1
while n > 1:
    factorial *= n
    n -= 1                # the step toward exiting
print(factorial)          # 120
```

The **trace table** is the course’s checking habit for loops: follow the variables round by round, on paper first. For the factorial loop: $(n=5, f=1) \rightarrow (4, 5) \rightarrow (3, 20) \rightarrow (2, 60) \rightarrow (1, 120)$. Build the table by hand, then add a `print(n, factorial)` inside the loop and let Python grade you.

Worked example 2 (a search with `break`, escalated). Find the first multiple of 7 that is greater than 100 – the trip count is unknown, the condition decides:

```
n = 100
while True:
    if n % 7 == 0:
        break                # found it - leave the loop now
    n += 1
print(n)                    # 105
```

The trace is short: 100, 101, ..., 104 all fail the test; 105 passes and `break` exits. `break` leaves the loop early; `continue` skips to the next round. Both are readability tools – a loop that must scan until it finds something, then stop, reads best with `break`. The further escalation, doubling money: at 5% a year, how long does 1000 take to reach 2000?

```
balance = 1000.0
years = 0
while balance < 2000.0:      # the condition itself is the goal
    balance *= 1.05
    years += 1
print(years, round(balance, 2)) # 15 2078.93
```

Fifteen years, and the loop stops the moment the condition flips – the trace table would show balance crossing the line at 2078.93, not landing on 2000 exactly.

Common mistakes. Editing the list (unit 1.5) or the loop variable you are `for`-looping over. `range(1, 100)` when 100 was meant – the end is excluded. A `while` whose exit step is missing or commented out: the condition never flips, and the program freezes.

Summary: `for` for known trip counts, `while` for conditions; every loop shows start, condition and step; trace on paper, then print-grade; `break` and `continue` for clean early exits.

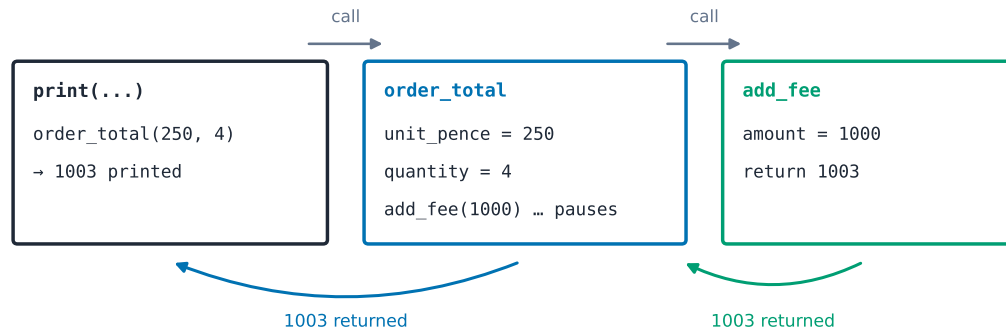
Functions: naming a job once

The same price calculation appears in five places; the tax rule changes; four of the five get updated. A **function** packages a job behind a name so it is written once and trusted everywhere: parameters in, a **return value** out. `def` defines; calling runs.

```
def total_price(count, unit=3.50):
    """Return the total for count items at unit price."""
    return count * unit

print(total_price(6))           # 21.0
print(total_price(6, unit=4.0)) # 24.0
```

The definition gives a **parameter** (count, unit); the call supplies an **argument** (6). A default value (unit=3.50) makes an argument optional. `return` hands a value back and ends the call; **printing is not returning** – a function without `return` returns `None`, and the caller that tries arithmetic on it gets a `TypeError`.



Nested calls: the caller pauses until the value comes back

Each call gets its own **local scope**: names inside are invisible outside, and disappear when the call ends. A call trace of nested calls reads: caller pauses → callee runs → return value lands in the caller.

Worked example 1 (the fee-inclusive order).

```
def add_fee(amount):
    """Return amount plus a fixed 3 pence fee."""
    return amount + 3

def order_total(unit_pence, quantity):
    """Return the order total: subtotal plus the fixed fee."""
    return add_fee(unit_pence * quantity)

print(order_total(250, 4)) # 1003
```

The contract in the docstring matters: whole pence in, whole pence out. Tracing `order_total(250, 4)`: subtotal 1000, `add_fee` returns 1003 – and the frame diagram above shows the pause and the hand-back.

Worked example 2 (a real algorithm inside a small function, escalated). `is_prime(n)` returns `True` when nothing from 2 up to $\sqrt{n}$ divides `n`:

```
def is_prime(n):
    if n < 2:
        return False
    d = 2
    while d * d <= n:           # only test up to the square root
        if n % d == 0:         # found a divisor - answer known, stop
            return False
        d += 1
    return True

print([p for p in range(2, 30) if is_prime(p)])
# [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Two function habits meet here: an early `return` ends the call the moment the answer is known (no flag variable needed), and the loop bound `d * d <= n` comes from the derivation – if $n = a \times b$ with $a \leq b$, then $a \leq \sqrt{n}$, so testing beyond $\sqrt{n}$ re-tests divisors already seen. Run it: the list is exactly the primes under 30.

Common mistakes. Printing inside a function instead of returning, so the result cannot be reused. Forgetting the `return` on one branch – that path quietly yields `None`. Reaching into a function's locals from outside; they are gone when the call ends.

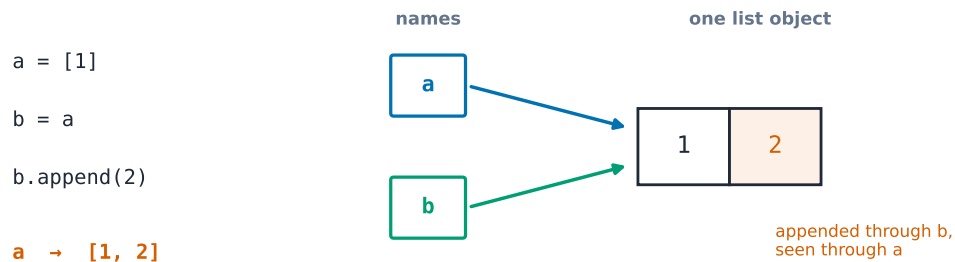
Summary: `def` once, call anywhere; parameters in, one value out via `return`; early returns end the call; locals live and die with the call; docstrings state the contract.

Collections: lists, dictionaries, tuples, sets

A café's till needs the day's orders in order, the menu by name, a location that must not drift, and today's unique drinks. Four jobs, four containers – choosing among them is the design decision of unit 1.5.

A **list** is an ordered, changeable sequence: `marks = [72, 85, 61]`, indexed from 0, sliced with `marks[1:3]`. A **dictionary** maps keys to values: `price = {"flat white": 3.5, "tea": 2.8}`, read and written by key. A **tuple** is a fixed sequence (53.1, -1.5) – good for coordinates that should not change. A **set** holds unique items with no order: `{"a", "b"}`.

```
marks = [72, 85, 61]
marks.append(90)           # [72, 85, 61, 90]
best = max(marks)           # 90
average = sum(marks) / len(marks) # 77.0
```



Aliasing: two names, one list

Assignment never copies: `a = [1]; b = a; b.append(2)` makes `a [1, 2]` – the list is shared, the names are not. This **aliasing** is unit 1.1's boxes again, now with two tags on one box, and it explains the next rule too: mutating a list while looping over it skips elements – loop over a copy (`for m in marks[:]`) or build a new list.

Worked example 1 (tally by category).

```
orders = ["tea", "flat white", "tea", "tea", "mocha"]
tally = {}
for item in orders:
    tally[item] = tally.get(item, 0) + 1
print(tally) # {'tea': 3, 'flat white': 1, 'mocha': 1}
```

The dictionary's `.get(key, 0)` supplies 0 for a first sighting – the idiom that makes counting a one-liner. Verify by hand: "tea" appears three times in `orders`, and the printed tally says so.

Worked example 2 (the day's best seller, escalated). Building on the tally, report the top item and its count:

```
best = max(tally, key=tally.get)
print(best, tally[best]) # tea 3
```

The `key=tally.get` argument asks `max` to compare items by their counts rather than alphabetically – the same `key=` trick ranks any collection by any measure. The escalation matters: real programs chain these tools (count, then rank) rather than re-scanning `orders` twice.

Choosing the collection is the design decision: order + duplicates → list; lookup by name → dictionary; fixed grouping → tuple; uniqueness → set.

Common mistakes. `tally[item] += 1` on a key that is not yet in the dictionary (`KeyError`) – `.get` is the fix. Expecting a set or dictionary to remember order. Assuming `b = a` copies a list; it aliases one.

Summary: four containers for four jobs; assignment shares, never copies; `.get` makes counting safe; `key=` makes ranking a one-liner.

Files and errors

The café closes, the power cuts – and the morning's tally survives only if the program wrote it to disk. A **file** outlives the program. `with open(path) as f:` opens, closes on exit, and reads text; `open(path, "w")` writes (and empties first – the classic accident).

```
with open("log.txt", "w") as f:
    f.write("opened\n")

with open("log.txt") as f:
    for line in f:
        print(line.rstrip()) # rstrip removes the trailing newline
```

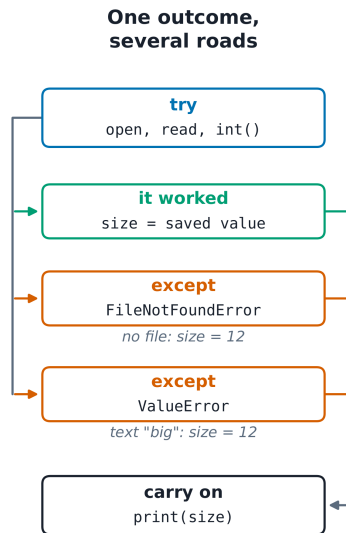
The `with` block is the safety rail: whatever happens inside it, the file closes. Reading gives lines *with* their newline, so `.rstrip()` belongs in every loop.

Errors are values Python raises as **exceptions**: `FileNotFoundError`, `ValueError`, `KeyError`, `TypeError`. `try/except` catches a named type – never a bare `except`: that hides real bugs.

Worked example 1 (the absent settings file). Reading a text size from a settings file that may be missing, or may not hold a whole number:

```
try:
    with open("settings.txt") as f:
        size = int(f.read().strip())
except FileNotFoundError:
    size = 12                # no file yet: a sensible default
except ValueError:
    size = 12                # the file holds "big", not a number
print("Text size", size)
```

Run it three times – with 16 in the file, with no file, and with **big** in the file – and it prints a usable size each time, from three different routes. That is the shape of robust code: one sensible outcome, several roads to it. Handle the error you expect, where you can respond to it; let the rest crash loudly.



Worked example 2 (a robust input loop, escalated). Ask until the user types a whole number:

```
def read_count():
    while True:
        try:
            return int(input("How many? "))
        except ValueError:
            print("Whole numbers only.")

print(read_count())
```

Feed it **x** then **4**: the first trip raises, is caught, prints the message, and the **while True** asks again; the second trip's **int(...)** succeeds and **return** exits the loop *and* the function in one move. Three units combine here – the loop (1.3), the function contract (1.4) and the exception (1.6) – which is the point of the escalation.

Common mistakes. `open(path, "w")` on the wrong path: the file is created or emptied before you notice. Catching **Exception** broadly and continuing with broken state. Forgetting `.rstrip()`, so output shows double-spaced lines.

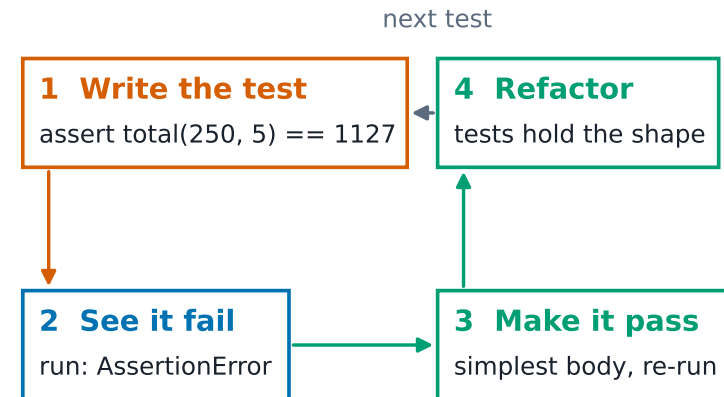
Summary: `with open(...)` for both reading and writing; "w" empties first; exceptions are values – catch the named type you expect, fall back sensibly, let the rest crash.

Decomposition, testing and debugging

Two teams write the same app; one ships in a week, one drowns for a month. The difference is not typing speed – it is that the fast team's code is built from small pieces they can each prove correct while the slow team debugs a whole app at once. Unit 1.7 is that discipline.

Decomposition splits a problem into functions of one job each – each small enough to test alone. **Testing** compares a function's actual output with the expected, at the boundaries: zero, one, empty, negative, huge.

Red, green, refactor



The test-driven loop: red, green, refactor

```
def add_fee(amount):
    return amount + 3

def order_total(unit_pence, quantity):
    return add_fee(unit_pence * quantity)

assert order_total(250, 4) == 1003
assert order_total(250, 0) == 3      # the zero-quantity boundary
assert order_total(0, 4) == 3
```

An `assert` that never fires is a passing test. Write the tests before you write the body when you can – they are the contract made executable, and the loop above (red → green → refactor) is that habit running continuously.

Worked example 1 (grow the tests with the function). Extend `order_total` with a 10% discount over £10 – the test is written first, at the new boundary:

```
def order_total(unit_pence, quantity):
    subtotal = add_fee(unit_pence * quantity)
    if subtotal >= 1000:          # pence: over £10
        subtotal = int(subtotal * 0.9)
    return subtotal

assert order_total(200, 4) == 803    # under £10 - unchanged
assert order_total(250, 4) == 902    # 1003 pence → 10% off → 902.7 → 902
assert order_total(250, 5) == 1127   # 1253 pence → 1127.7 → 1127
```

Check the middle test by hand: subtotal 1003, ten percent off gives 902.7, and `int(...)` rounds down to 902. Notice the first assert too: at 200 pence and four cups the total stays 803 – the boundary tests prove both that the discount fires and that it does not fire early.

Worked example 2 (bisect a bug with prints, escalated). The tally from 1.5 prints `{'tea': 3}` but the best-seller line prints the wrong drink. The method: (1) reproduce with the smallest failing input; (2) print the values at the boundary between the trusted part and the suspected part – here, the tally before the `max`; (3) fix one thing; (4) re-run the tests. Moving the print halfway through the code each run halves the search: four prints find a bug in a sixteen-line program, however well it hides.

Debugging is finding the difference between what you believe the code does and what it does – the print at the trust/suspect boundary is the instrument.

Common mistakes. Testing only the happy path: the boundaries (zero, one, empty, negative) are where programs break. Writing a week of code before the first test, then debugging a mountain. Fixing two things at once, so neither result means anything.

Summary: decompose until every piece tests alone; assert the boundaries the minute

a function exists; write the test first when you can; debug by bisection with prints at trust boundaries – the habit the whole degree builds on.