CSSE1001 — Week 3 Notes

Table of contents

Functions **2**
- Outline . . . 2
- Learning objectives . . . 2
- What is a function? . . . 2
- Building up Heron's formula . . . 3
- `return`, and what it does to the lines below it . . . 5
- Exercise: the middle number . . . 6
- Two worked questions . . . 7
- Style: PEP8 . . . 7
- Summary . . . 8
- Next lecture . . . 8

Sequence, Selection, and Iteration **8**
- Outline . . . 8
- Learning objectives . . . 8
- Part 1 — logic . . . 8
- Part 2 — the `if` statement . . . 10
- Part 3 — factoring and refactoring . . . 12
- Summary . . . 13
- Next lecture . . . 13

Reference material **14**
- Python Boolean Logic . . . 14
- Python Functions . . . 19
- Python If Statements . . . 26
- Python PEP8 Style Guide . . . 36
- Python Primitive Data Types . . . 38
- Python Testing (doctest & Assertions) . . . 50

Functions

Lecture 3A. See csse1001[1] for course logistics.

Outline

- What a function is and how to define one
- The `return` statement, and how it differs from `print`
- Improving a function with type hints and docstrings
- PEP8: how the course expects that code to be laid out

Learning objectives

1. User-defined functions bundle code together and are the building blocks of more sophisticated programs.
2. `return` hands back a value **and exits the function**; it is not the same as `print`.
3. Functions can be documented with type hints and docstrings, neither of which Python enforces.

What is a function?

User-defined functions are perhaps the single most important feature of programming languages. They let us **abstract away** complexity by **bundling** lines of code together, and are themselves the building blocks of more sophisticated functions.

We have *already* been using functions — `*` and `max`, for example. Each takes **input** and returns **output**: multiplication takes two numbers and returns a number; `max` takes two or more numbers and returns a number.

Python functions are analogous to those of mathematics and can also be named. A parabola in mathematics is given by the function $f(x) = x^2 + x + 1$, and in Python we write

```
>>> def f(x):
...     return x**2 + x + 1
>>> f(3)
13
```

[1] courses/csse1001/csse1001.pdf

See python-functions[2] for the full syntax rules — indentation, multiple parameters, the `return` statement, `return` vs. `print`, type hints, and docstrings — all of which are used throughout the rest of this note.

Building up Heron's formula

One function, built three times over, each version better than the last. The exercise:

> **Exercise (Heron's formula).** The area of $\triangle ABC$ is $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{1}{2}(a+b+c)$. *Write a function which computes the area of a triangle from the side lengths a, b, c.*

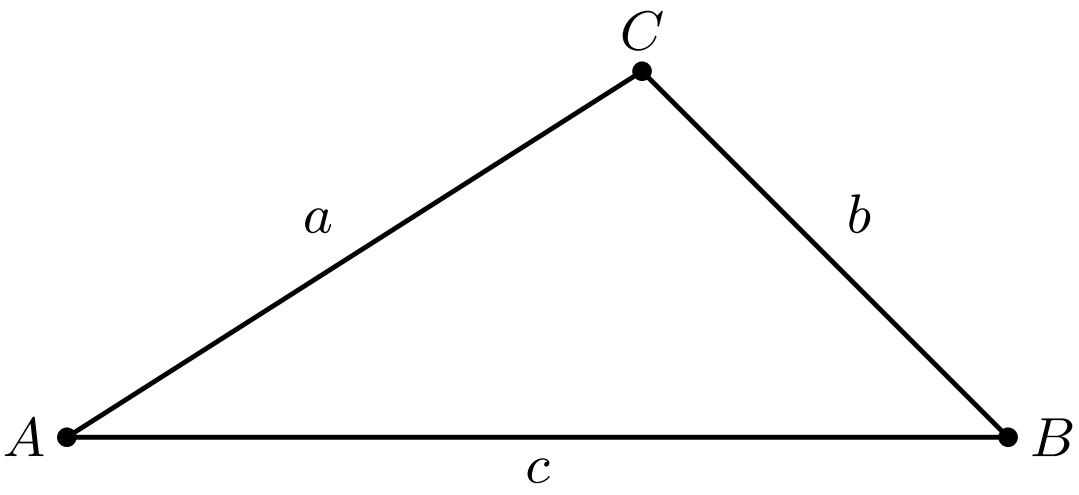

Figure 1: The triangle and its side labels: a on the side from A to C, b on the side from C to B, c on the side from A to B. That is not the usual convention, under which each side is named for the vertex opposite it — here only c is opposite its namesake vertex, while a sits alongside A and b alongside B. Heron's formula is symmetric in a, b and c so this changes no arithmetic, but it changes what you should picture when the code says a.

Question. What is the area of the triangle with sides 3, 4, and 5?

Answer 1 — Python as a calculator. Everything inline, `s` spelled out four times over:

```
>>> (0.5*(3+4+5)
```

[2] courses/csse1001/python-functions.pdf

```
... *(0.5*(3+4+5)-3)
... *(0.5*(3+4+5)-4)
... *(0.5*(3+4+5)-5))**0.5
```

We have to type a *lot*. It would be hard to repeat.

Answer 2 — using names and sequencing. Naming `s` once removes the repetition (see 2025-08-04-python-memory-model[3]):

```
>>> a, b, c = 3, 4, 5
>>> s = (a + b + c)/2
>>> (s*(s-a)*(s-b)*(s-c))**0.5
6.0
```

But the names are now stuck at those three values — a different triangle means retyping the lot.

Answer 3 — using functions. The same three lines, parameterised and reusable for *any* triangle:

```
>>> def heron(a, b, c):
...     s = (a + b + c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5

>>> heron(3, 4, 5)
6.0
```

The result is `6.0`, not `6`: `**0.5` produces a float.

Improving the function: type hints

In mathematics we would define a mapping

$$\text{triangle_area} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}.$$

In Python we can write that mapping into the `def` line itself:

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
>>> triangle_area(2.2, 3.3, 4.4)
3.5147323866832307
```

[3] courses/csse1001/2025-08-04-python-memory-model.pdf

Type hints are **not enforced**. They are documentation, for making the code more readable — annotate the same function `str` throughout and it still returns the same float. See python-functions[4].

Improving the function: docstrings and examples

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     """
...     Return the area of the triangle with sides length <a>,
...     <b>, and <c>.
...     Preconditions: <a>, <b>, <c> are all nonzero positive.
...     >>> triangle_area(3, 4, 5)
...     6.0
...     """
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
```

That is the full shape the course expects of a Python function: `def` line with type hints, docstring with a short description, preconditions, and a worked example, then the body, then `return`.

`return`, and what it does to the lines below it

`return` is a **reserved word** — one that cannot be assigned by us. It designates what value the function should return *while also terminating the function itself.*

The striking case is the one where the `return` is the *first* line:

```
>>> def example(x):
...     return 3*x
...     return 2*x
...     print(1*x)

>>> a = example(1)
>>> a
3
```

[4] courses/csse1001/python-functions.pdf

Neither the second `return` nor the `print` ever runs. The other variants of this question, and the diagram of what control actually does, are in python-functions[5].

A function with **no** `return` at all returns `None`:

```
>>> def example(x):
...     print(2*x)
>>> a = example(1)
2
>>> a
>>> type(a)
<class 'NoneType'>
```

The default return value is `None`, and `print` is exactly this kind of function: it returns nothing, and its *side effect* is that it displays something.

Exercise: the middle number

> **Exercise.** Write a function that takes three integers and returns the number which is **not** the largest or the smallest.

```
def middle_number(x: int, y: int, z: int) -> int:
    """Return the number that is not the largest or smallest
    among the three inputs.

    Precondition: The numbers are distinct.

    >>> middle_number(3, 1, 2)
    2
    >>> middle_number(2, 3, 1)
    2
    """
    return (x+y+z) - min(x, y, z) - max(x, y, z)
```

The trick is that the three numbers sum to (smallest + middle + largest), so subtracting `min` and `max` leaves the middle one — no comparisons or `if`-statements needed.

[5] courses/csse1001/python-functions.pdf

Two worked questions

What is the type of `foo`'s return value?

```
def foo(radius: int):
    """
    Precondition:  radius > 0
    """
    area = 3.14159 * radius**2
```

`NoneType`. There is no explicit `return` statement, so the computed area is thrown away.

Is this valid Python code?

```
def foo(x: int, y: int) -> int:
    """
    >>> foo(2, 3)
    6
    """
    return x
ans = foo("oi", 3)
```

Yes. Type hints are *not* enforced. The docstring's own example is wrong too — this `foo` returns `x`, so `foo(2, 3)` is `2`, not `6` — and Python will not complain about that either unless the doctests are actually run. See python-testing[6].

Style: PEP8

The style rules apply to every function above. A **PEP** is a Python Enhancement Proposal — a design document on how to *style* our code — and this course follows Google's PEP, a variant of PEP8. In brief: names start with a letter, are lowercase and underscore-separated; lines stay strictly under 80 characters, wrapped inside a redundant enclosing bracket; and a comment should explain something not obvious in the design rather than restate the line it sits on. The yes/no tables for naming and spacing, and the rule behind the spacing one, are in python-pep8-style-guide[7].

[6] courses/csse1001/python-testing.pdf
[7] courses/csse1001/python-pep8-style-guide.pdf

Summary

Blocks of code can be grouped into **functions**. Functions take zero or more inputs and hand back **a single value** designated by `return`.

Next lecture

2025-08-11-sequence-selection-and-iteration[8] — selection.

Sequence, Selection, and Iteration

Lecture 3B. Course logistics are in csse1001[9].

Outline

Despite the title, this is entirely about **selection** — *controlling program flow.* Sequencing is in 2025-08-04-python-memory-model[10], and iteration is the topic of the next lecture. The order is: boolean logic → the `if` statement → refactoring the if-statements you end up with.

Learning objectives

1. Predicates and the logical operators `and`, `or`, `not` let us build up complex conditions.
2. The `if` statement, and its `elif`/`else` variants, lets us skip or select blocks of code based on a condition.
3. If-statements can often be refactored into simpler, equivalent forms — and some innocuous-looking ones are silently wrong.

Part 1 — logic

The basic operands of logic are `True`, `False`, `or`, `and`, `not`. Let $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$ be the **boolean domain**; a **predicate** is any function mapping into $\mathbb{B}$, and `and`, `or` are themselves predicates $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$.

Three results that catch people out:

[8] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf
[9] courses/csse1001/csse1001.pdf
[10] courses/csse1001/2025-08-04-python-memory-model.pdf

- **`and` binds tighter than `or`**, and `not` is evaluated first among the three. `not` is still *looser* than the comparison operators, so `not a == 6` parses as `not (a == 6)`.
- **`and`/`or` short-circuit**, so `True or 1/0` is `True` while `False or 1/0` raises. Short-circuiting saves you from *evaluating* the right operand, not from *parsing* it: `True or !` is still a `SyntaxError`.
- **`and`/`or` do not return `True`/`False`** — `and` returns its first falsy operand, `or` its first truthy one, and the last operand if there is no such value. Hence `0 or 2` is `2`.

Full reference, truth tables, and worked transcripts: python-boolean-logic[11].

Worked question — truthiness

Which of these evaluate to `True`?

```
bool(not () and "hello")
bool(True or 1/0)
bool(" ")
bool(2 > True)
bool("False")
```

All five. `()` is falsy, so `not ()` is `True` and `and` hands back `"hello"`; `or` short-circuits before dividing; `" "` is a non-empty string; `True` is `1`, so `2 > True`; and `"False"` is a non-empty string whatever it spells.

Exercises

Bracket for `False`. Bracket the expression below so that it evaluates to `False`. How many different bracketings can you find?

```
False and False or True and False or False or True
```

Add a single `not`. Add a single `not` to the same expression so that it evaluates to `False`.

[11] courses/csse1001/python-boolean-logic.pdf

Part 2 — the `if` statement

> **Definition (if-statement).** Given a **condition** C (i.e. a predicate that evaluates to `True` or `False`), an **if-statement** is a **control structure** that executes a block of code when C is `True` (and skips it otherwise).

It comes in four shapes — `if`, `if/else`, `if/elif`, `if/elif/else` — each with its own flowchart. The four flowcharts, plus the code templates and the warnings that go with them, are in python-if-statements[12].

Two warnings:

```
>>> if False:
...     ans = 0
>>> ans
NameError: name 'ans' is not defined
```

A variable assigned only inside a skipped block does not exist afterwards. And:

```
>>> if x == 1 or 2 or 3:
...     print("hello")
```

This **always** prints, because it parses as `(x == 1) or 2 or 3` and `2` is truthy on its own. You need `x == 1 or x == 2 or x == 3` — later refactored to `x in [1,2,3]`.

Worked example — the beverage chain

The running example, and the bug it is built to expose:

```
>>> age = 60

>>> if age >= 18:
...     beverage = "cheap beer"
... elif age >= 30:
...     beverage = "standard beer"
... elif age >= 50:
...     beverage = "expensive beer"

>>> beverage
'cheap beer'
```

[12] courses/csse1001/python-if-statements.pdf

This code does not work as intended. Nothing in it is a syntax error and every condition is individually correct; the fault is entirely in the **order**. Because each `elif` is only reached when everything above it failed, and `age >= 18` is the *loosest* test, a 60-year-old is caught by the first branch and the other two can never fire for anybody:

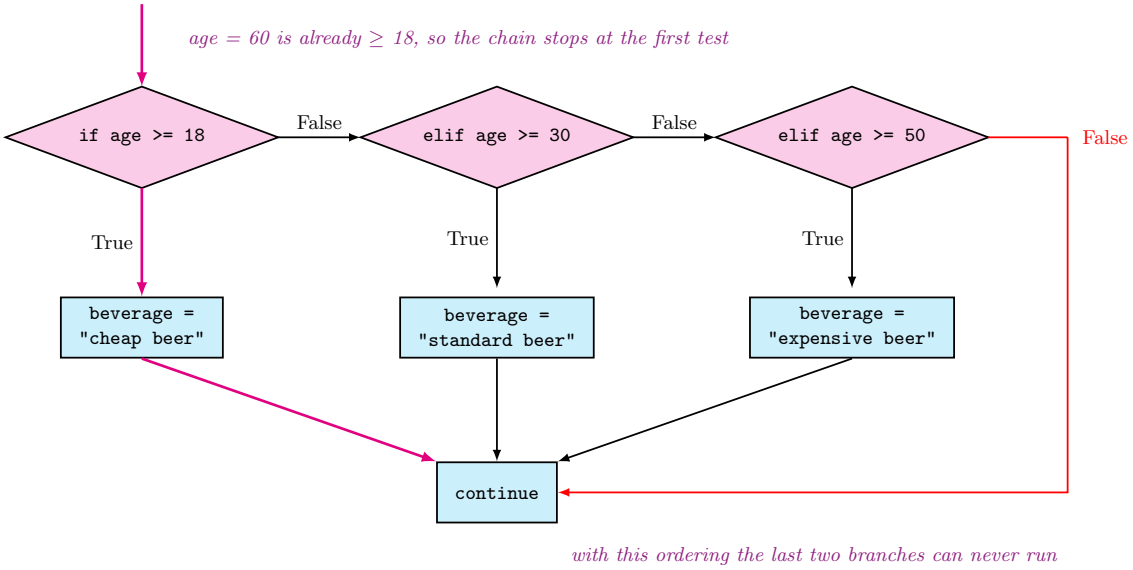

Figure 2: The buggy beverage chain as an if/elif/elif flowchart. The magenta trace is the path age = 60 takes. The red edge is the False exit from the last test, where no branch runs. Because age >= 18 is the loosest test and sits first, any age reaching the second diamond is already under 18 — so the second and third branches can never run, for any age at all.

Fix 1 — make the ranges disjoint (`18 <= age < 30`, `30 <= age < 50`, `50 <= age`). Once they are disjoint the `elif`s are not earning their keep; three separate `if`s would do the same job.

Fix 2 — keep the open-ended tests but order them tightest-first (`>= 50`, then `>= 30`, then `>= 18`), so each `elif` inherits a guarantee from the ones above it: reaching the second test guarantees `age < 50`, reaching the third guarantees `age < 50 and age < 30`.

Both fixes still leave `age = 9` with no branch at all and hence a `NameError` on `beverage`; adding `else: beverage = "apple juice"` is what closes that off. The code for all of these, and the flowchart contrast between a chain with and without an `else`, is in python-if-statements[13].

[13] courses/csse1001/python-if-statements.pdf

Part 3 — factoring and refactoring

> **Definition (factoring).** In computer science **factoring** means breaking a complex problem into parts that are easier to conceive, understand, program, and maintain.
>
> **Definition (refactoring).** Code **refactoring** is the process of restructuring existing computer code — changing the factoring — without changing its behaviour.

The if-statement simplifications (nested `if`s → one `and`ed condition; `if/else` returning `True`/`False` → return the condition; `== True` / `== False` → drop it or `not` it) are written out in python-if-statements[14].

Worked question — is `elif` the same as a second `if`?

Equivalent. An `elif` may drop whatever the condition above it already ruled out:

```
>>> if x > 0:                      >>> if x > 0:
...     print("A")                  ...     print("A")
... elif x <= 0 and x % 2 == 0:     ... elif x % 2 == 0:
...     print("B")                  ...     print("B")
```

Reaching the `elif` already guarantees `x <= 0`, so the extra conjunct is redundant.

Not equivalent. Change both `elif`s to `if` and the same edit becomes a bug, because the second `if` is independent of the one before it:

```
>>> if x > 0:                      >>> if x > 0:
...     print("A")                  ...     print("A")
>>> if x <= 0 and x % 2 == 0:      >>> if x % 2 == 0:
...     print("B")                  ...     print("B")
```

Let `x = 2`. The first prints `A`; the second prints `A` **and** `B`.

Refactoring exercise

> **Exercise.** Refactor the code.

[14] courses/csse1001/python-if-statements.pdf

```
def foo(x, y):
    if x > 100 and y > 0:
        if y > 100 and x > 0:
            return "A"
        elif y > 100 or x > 0:
            return "A"
        else:
            return "B"
    elif y <= 0 or x <= 0:
        if x == y:
            return "A"
        if x <= y and x >= y:
            return "B"
        if y < x and x < y:
            return "C"
        else:
            return "A"
    else:
        if x <= 100 and y > 0:
            return "A"
        if x > 100 or y <= 0:
            return "B"
        else:
            return "D"
```

The tools it is built to exercise are all above: conditions that are already guaranteed by an enclosing branch, mutually exclusive conditions that can never both hold (`x <= y and x >= y` means `x == y`; `y < x and x < y` is impossible), and `if`/`return` pairs that need no `else`.

Summary

Blocks of code can be **skipped** using `if` statements. This **control flow** depends on the evaluation of **predicate** statements.

Next lecture

2025-08-18-while-loops[15] — iteration.

[15] courses/csse1001/2025-08-18-while-loops.pdf

Reference material

Python Boolean Logic

The basic operands of logic are `True`, `False`, `or`, `and`, and `not`.

Boolean domain and predicates

> **Definition (boolean domain).** Let $\mathbb{B}$ denote the **boolean domain**, where $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$.
>
> **Definition (predicate).** Any function that maps into $\mathbb{B}$ — i.e. one that evaluates to `True` or `False` — is called a **predicate**.

So the boolean function $> : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ ("greater than") is a predicate:

```
>>> type(True), type(False)
(<class 'bool'>, <class 'bool'>)
>>> 7 > 3
True
>>> 7 >= 7 + 1          # note addition happened first
False
>>> (7 >= 7) + 1
2
>>> int(True), int(False)
(1, 0)
>>> bool(1), bool(0), bool(123), bool(-2)     # "truthiness"
(True, False, True, True)
```

The two middle lines are worth pausing on: arithmetic binds tighter than comparison, so `7 >= 7 + 1` is `7 >= 8`; but once a comparison *has* been evaluated its `True`/`False` result is an ordinary value that arithmetic will happily consume, so `(7 >= 7) + 1` is `True + 1` is `2`.

`and` and `or`

> **Definition (and).** The binary predicate `and` is used to express that **both** of two statements are true, and is false *if either is false.*

`and` : $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$

and	True	False
True	True	False
False	False	False

Definition (or). The binary predicate `or` is used to express that **at least one** of two statements is true, and is false *only when both are false*.

`or` : $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$

or	True	False
True	True	True
False	True	False

```
>>> 3 > 7 or 7 > 3
True
>>> 3 > 7 and 7 > 3
False
>>> 0 < 3 and 3 < 8
True
>>> 0 < 3 < 8          # shorthand for the above
True
```

Precedence

`True or False and False` is **ambiguous** without a precedence rule, because

```
(True or False) and False == False
True or (False and False) == True.
```

Observing what Python actually does settles it:

```
>>> True or False and False
True
```

and thereby **`and` has precedence over `or`**.

Definition (not). The logical statement `not` is the negation of a logical statement: `not True == False` and `not False == True`. It is evaluated *first* when combined with other logical operands.

"First" here means first among the *logical* operands — `not` binds tighter than `and` and `or`, but **looser than the comparison operators**, so a `not` in front of a comparison negates the whole comparison rather than just its left operand:

```
>>> a, b = 6, 7

>>> a == 6 and b != 5
True

>>> not (a == 6 and b != 5)
False

>>> not a == 6 and b != 5      # not happens before and
False

>>> not (a != 6 or not b != 5) # avoid double negations
True
```

The third line is `(not (a == 6)) and (b != 5)`, i.e. `False and True`. The fourth is valid and correct but is exactly the kind of double negation to avoid — it is far harder to read than the first line it is equivalent to.

The full ladder for these three, tightest first: comparisons (`==`, `!=`, `<`, …) → `not` → `and` → `or`. Arithmetic binds tighter than all of them; see python-operator-precedence-and-associativity[16]. The transcript above does not settle where `not` sits relative to the comparisons, because `(not a) == 6` and `not (a == 6)` happen to give the same answer on its third line.

Short circuits (lazy evaluation)

`or` stops as soon as it finds a truthy value and `and` stops as soon as it finds a falsy one — the remaining operand is then never evaluated, so an error lurking in it never fires:

```
>>> True or 1/0
True
>>> False or 1/0
ZeroDivisionError: division by zero
>>> True or non_existent_variable
True
```

[16] courses/csse1001/python-operator-precedence-and-associativity.pdf

Python does not bother looking for `non_existent_variable` at all. What short-circuiting does *not* buy you is a pass on being well-formed — the expression still has to parse:

```
>>> True or !
  True or !
         ^
SyntaxError: invalid syntax
```

The `and` cases are the mirror image:

```
>>> True and 1/0
ZeroDivisionError: division by zero
>>> False and 1/0
False
```

Comparisons are still expressions

A comparison produces a value like any other expression, and that value can be named:

```
>>> a = 3
>>> a == 3
True
>>> a != 3
False

>>> b = (a != 3)     # comparisons are still expressions
>>> b
False
```

Because `True` and `False` have integer representations 1 and 0, they also survive arithmetic. In the example below, `a` is `True + 2` = 3, `b` is `7*False` = 0, and `c` is `not (a != b)` = `not (3 != 0)` = `not True`:

```
>>> a = True + 2
>>> a
3                    # True has an integer representation as 1
>>> b = 7*False
>>> b
0                    # False has an integer representation as 0
>>> c = not a != b
>>> c
False
```

Truthiness

Truthy values are values in Python that evaluate to true — for example non-zero numbers, non-empty strings, and non-empty lists:

```
>>> bool(1), bool(-10), bool("Hello"), bool([1,2,3])
(True, True, True, True)
```

Falsy values are values that evaluate to false — for example zero, the empty string, and the empty list:

```
>>> bool(0), bool(""), bool([])
(False, False, False)
```

Question. Which of the following expressions evaluate to `True`?

```
bool(not () and "hello")
bool(True or 1/0)
bool(" ")
bool(2 > True)
bool("False")
```

All five, each for a slightly different reason: `()` is falsy so `not ()` is `True` and `and` hands back `"hello"`, which is truthy; `or` short-circuits before the division; `" "` is a *non-empty* string (a space is a character); `True` is `1` and `2 > 1`; and `"False"` is a non-empty string, its contents notwithstanding.

What `and`/`or` actually return

The functions `and` and `or` do not always return `True` and `False`. **`and` returns its first *falsy* input and `or` its first *truthy* input; if no such value exists, the last input is returned.**

```
>>> 0 or 2
2
>>> 0 and 2
0
```

Because the empty tuple `()` is falsy and any non-empty tuple is truthy:

```
>>> () or (1,) or (1, 2)
(1,)
>>> (1, 2) or (1,) or ()
(1, 2)
>>> 1 and 2
2
>>> () and (1,) and (1, 2)
()
>>> (1, 2) and (1,) and ()
()
```

This is what makes the truthiness question above work out the way it does: `not () and "hello"` evaluates to the string `"hello"`, not to `True`, and only the outer `bool` turns it into `True`.

Python Functions

User-defined functions bundle lines of code together so they can be reused, abstracting away complexity. Like mathematical functions, they take input and return output (we've already used built-in functions this way, e.g. `*` and `max`). Multiplication takes two numbers and returns a number; `max` takes two or more numbers and returns a number.

Defining a function

A mathematical function such as $f(x) = x^2 + x + 1$ is written in Python as:

```
>>> def f(x):
...     return x**2 + x + 1
>>> f(3)
13
```

Functions can take multiple parameters, and a call is an expression like any other — so calls nest inside other calls and inside arithmetic:

```
>>> def f(x, y):
...     return x*y
>>> x = f(3, 2)
>>> x
6
>>> x = f( -f(5, 2) + 12, f(2, 3) )
```

```
>>> x
12
```

The inner calls are evaluated before the outer one, so the last line is `f(-10 + 12, 6)`, which is `f(2, 6)`, and so `12`.

Indentation

Four spaces of indentation are significant in Python — they associate a line of code with the control structure above it (here, the function body with its `def`). Get the indentation wrong and you get

```
IndentationError: unexpected indent
```

which is a formatting complaint, not a logic error.

The `return` statement

`return` is a **reserved word** — one that cannot be assigned by us. It designates the value the function hands back **and terminates the function itself**.

> **Definition (`return`).** The `return` statement causes a function to exit and hand back a value to its caller.

Two consequences follow:

1. Statements before the `return` still run (including their side effects).
2. Nothing after the first `return` that is actually reached ever runs — not a further `print`, not a second `return`.

Three variants of the same function show both. In the first, both `print`s are above the `return`, so both run:

```
>>> def example(x):
...     print(1*x)
...     print(2*x)
...     return 3*x

>>> a = example(1)
1
2
>>> a
3
```

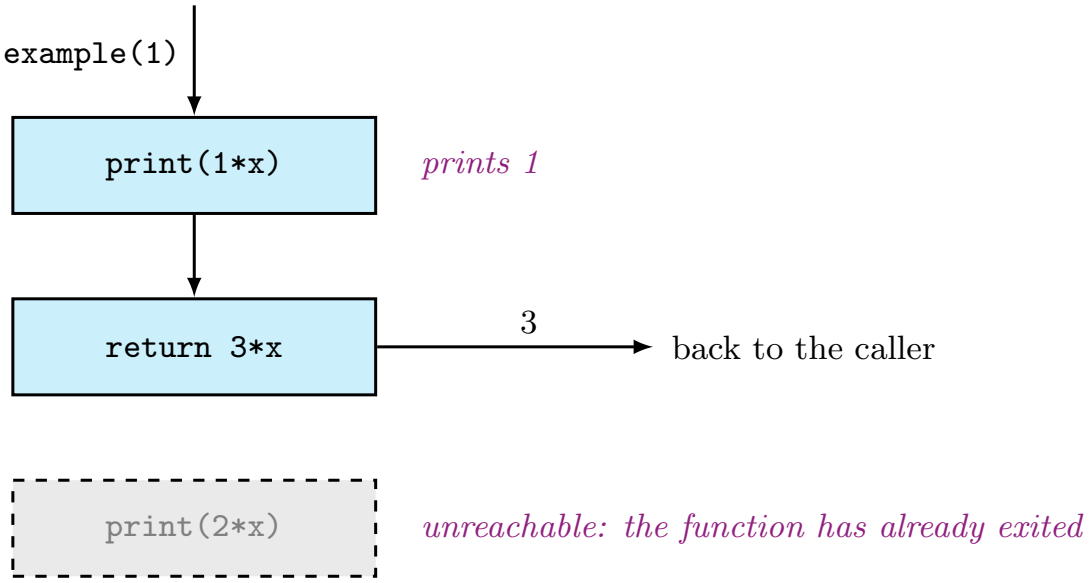

Figure 3: Control through the body of `def example(x): print(1*x); return 3*x; print(2*x)` when called as `example(1)`. The `print` above the `return` runs and displays 1; the `return` hands 3 back and ends the call there, so the `print` below it never executes. The dashed grey box marks that unreachable statement.

Move the `return` up one line and the second `print` is stranded below it:

```
>>> def example(x):
...     print(1*x)
...     return 3*x
...     print(2*x)

>>> a = example(1)
1
>>> a
3
```

Put the `return` first and *nothing* below it runs — not the second `return`, not the `print`. The call produces no output at all:

```
>>> def example(x):
...     return 3*x
...     return 2*x
...     print(1*x)

>>> a = example(1)
>>> a
3
```

If a function has no `return` statement at all, Python presumes `return None` is its last line:

```
>>> def f(x, y):
...     print(x+y)
...     return None      # presumed -- you do not write this
```

so the "no return" case is not special, it just returns `None`:

```
>>> def example(x):
...     print(2*x)
>>> a = example(1)
2
>>> a
>>> type(a)
<class 'NoneType'>
```

`>>> a` displays nothing at all, because the REPL does not echo `None`.

`return` versus `print`

`print` is a function that **returns nothing** and whose **side effect** is that it displays something:

```
>>> x = print("Hello World")
Hello World
>>> type(x)
<class 'NoneType'>
```

The distinction matters because `return` and `print` can look like they are doing the same thing — right up until the result is used in further computation:

```
>>> def f(x, y):
...     print(x+y)
>>> x = f(2, 3)
5
>>> y = f(3, 4)
7
>>> x + y
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'.
```

Compare with the use of `return`:

```
>>> def f(x, y):
...     return x + y
>>> x = f(2, 3)
>>> y = f(3, 4)
>>> x + y
12
```

`return` is a reserved word, **not a function** — so we write

```
>>> def f(x, y):
...     return x + y
```

and not

```
>>> def f(x, y):
...     return(x + y)
```

Type hints

Type hints annotate the expected type of each parameter and of the return value. In mathematics we would define a mapping triangle_area : $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$; in Python we write that as:

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
>>> triangle_area(2.2, 3.3, 4.4)
3.5147323866832307
```

Type hints are **not enforced** — they are documentation for making the code more readable, nothing more. Annotating the *same* function with `str` everywhere changes nothing about how it runs:

```
>>> def triangle_area(a: str, b: str, c: str) -> str:
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
>>> triangle_area(2.2, 3.3, 4.4)
3.5147323866832307
```

and so this is valid Python too, hints notwithstanding:

```
def foo(x: int, y: int) -> int:
    """
    >>> foo(2, 3)
    6
    """
    return x
ans = foo("oi", 3)
```

Docstrings

A **docstring** (`"""..."""` immediately under the `def` line) documents what a function does and its preconditions, and gives example calls (written like REPL input/output) that double as tests — see python-testing[17]:

[17] courses/csse1001/python-testing.pdf

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     """
...     Return the area of the triangle with sides length <a>,
...     <b>, and <c>.
...     Preconditions: <a>, <b>, <c> are all nonzero positive.
...     >>> triangle_area(3, 4, 5)
...     6.0
...     """
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
```

Inside a docstring the course refers to a parameter by its name in angle brackets, `<a>`.

General template

```
def function_name(arg0: type, arg1: type, ...) -> type:
    """
    Short description of function for documentation.
    Preconditions (if any).
    >>> function_name(x, y, ...)
    expected output
    """
    ...
    function body
    ...
    return
```

Worked question — a function that computes but never returns

A function that computes a value but never `return`s it still hands back `None`. What is the type of `foo`'s return value?

```
def foo(radius: int):
    """
    Precondition:  radius > 0
    """
    area = 3.14159 * radius**2
```

`NoneType`. There is no explicit `return` statement, so the computed `area` is simply discarded when the call ends.

See python-pep8-style-guide[18] for the naming, spacing, and line-length conventions used when writing functions like this.

Python If Statements

> **Definition (if-statement).** Given a condition C — i.e. a predicate that evaluates to `True` or `False`, see python-boolean-logic[19] — an **if-statement** is a **control structure** that executes a block of code when C is `True` (and skips it otherwise).

> **About the flowcharts below**
>
> The four flowcharts use one vocabulary: a **pale pink diamond** for the condition, a **pale blue rectangle** for a block of code, and black arrows labelled `True` / `False`.
> Red is used on only two of the four. In the two chain figures the *last* `False` edge — the one taken when every condition has failed — is red, arrow and label both. The single-`if` bypass and the `else` edge of a plain if-else are the same "the condition was false" edge and are black. So red marks the end of a chain here, not falsity in general.

If-then

```
if <cond>:
    <code executed when cond == True>
    ...
```

In Python **spaces matter** — only code indented within the if-statement gets executed. Everything else carries on regardless.

Because of truthiness[20], the condition doesn't need to literally be `True`/`False`. The two pairs below are the same code with a literal and then with a variable in the condition slot:

```
>>> x = 0
>>> if False:
...     x = x + 1
>>> x
0
```

```
>>> x = 0
>>> if x:
...     x = x + 1
>>> x
0
```

```
>>> x = 0
>>> if True:
```

```
>>> x = 1
>>> if x:
```

[18] courses/csse1001/python-pep8-style-guide.pdf
[19] courses/csse1001/python-boolean-logic.pdf
[20] courses/csse1001/python-boolean-logic.pdf

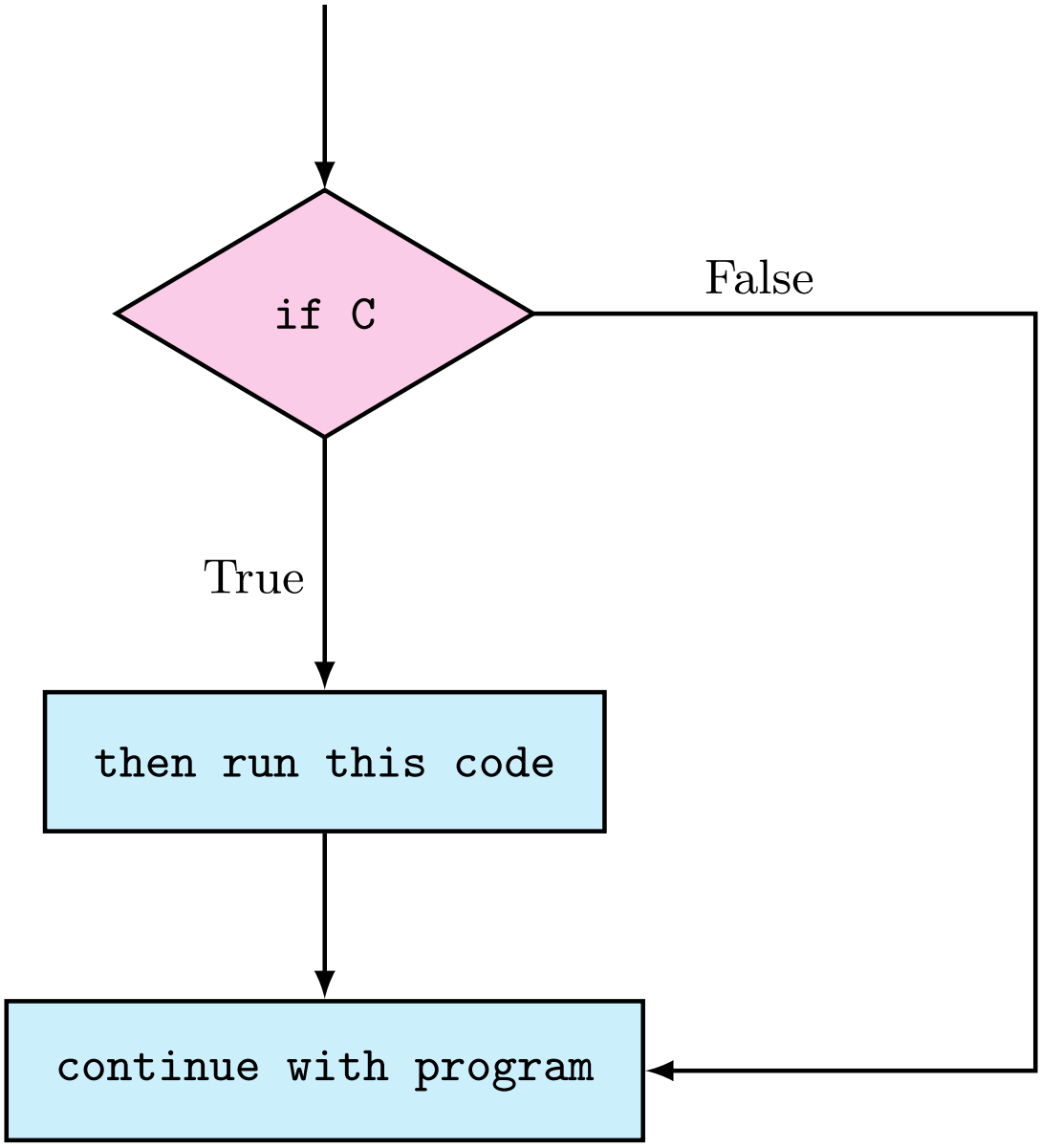

Figure 4: The if-then control structure. The True branch drops into the guarded block; the False branch bypasses it entirely and rejoins the program at the same place, which is why an if-statement can only ever *skip* code, never choose between alternatives.

```
...     x = x + 1     ...     x = x + 1
>>> x                >>> x
1                    2
```

Warning. A variable only assigned inside an if-block does not exist if the condition was false — the code in the if-statement is skipped, and therefore `ans` never gets set:

```
>>> if False:
...     ans = 0
>>> ans
NameError: name 'ans' is not defined
```

Two ifs are not two branches

Successive independent `if`s both get tested, so more than one of them can fire. With `print` that means two lines of output:

```
>>> def foo(x):
...     if x > 0:
...         print("Positive")
...     if x > 10**5:
...         print("Large positive")

>>> ans = foo(10**6)
Positive
Large positive

>>> type(ans)
<class 'NoneType'>
```

Swap the `print`s for `return`s and the behaviour changes completely, because `return` exits the function on the spot (see python-functions[21]) — the second `if` is never even reached:

```
>>> def bar(x):
...     if x > 0:
...         return "Positive"
...     if x > 10**5:
...         return "Large positive"
```

[21] courses/csse1001/python-functions.pdf

```
>>> ans = bar(10**6)
>>> type(ans)
<class 'str'>
>>> ans
'Positive'
```

If-then-else

Writing two complementary `if`s makes the code longer *and* checks the condition twice:

```
>>> if balance >= 0:
...     in_the_black = True
...     in_the_red = False

>>> if balance < 0:
...     in_the_black = False
...     int_the_red = True
```

`else` collapses that into one test:

```
if <cond>:
    <code>
else:
    <code>
```

```
>>> if balance >= 0:
...     in_the_black = True
...     in_the_red = False
... else:
...     in_the_black = False
...     int_the_red = True
```

`if-else` **does not allow you to skip code** — it lets you pick between two instruction sets.

`int_the_red` on the last line of both versions is a typo for `in_the_red`.

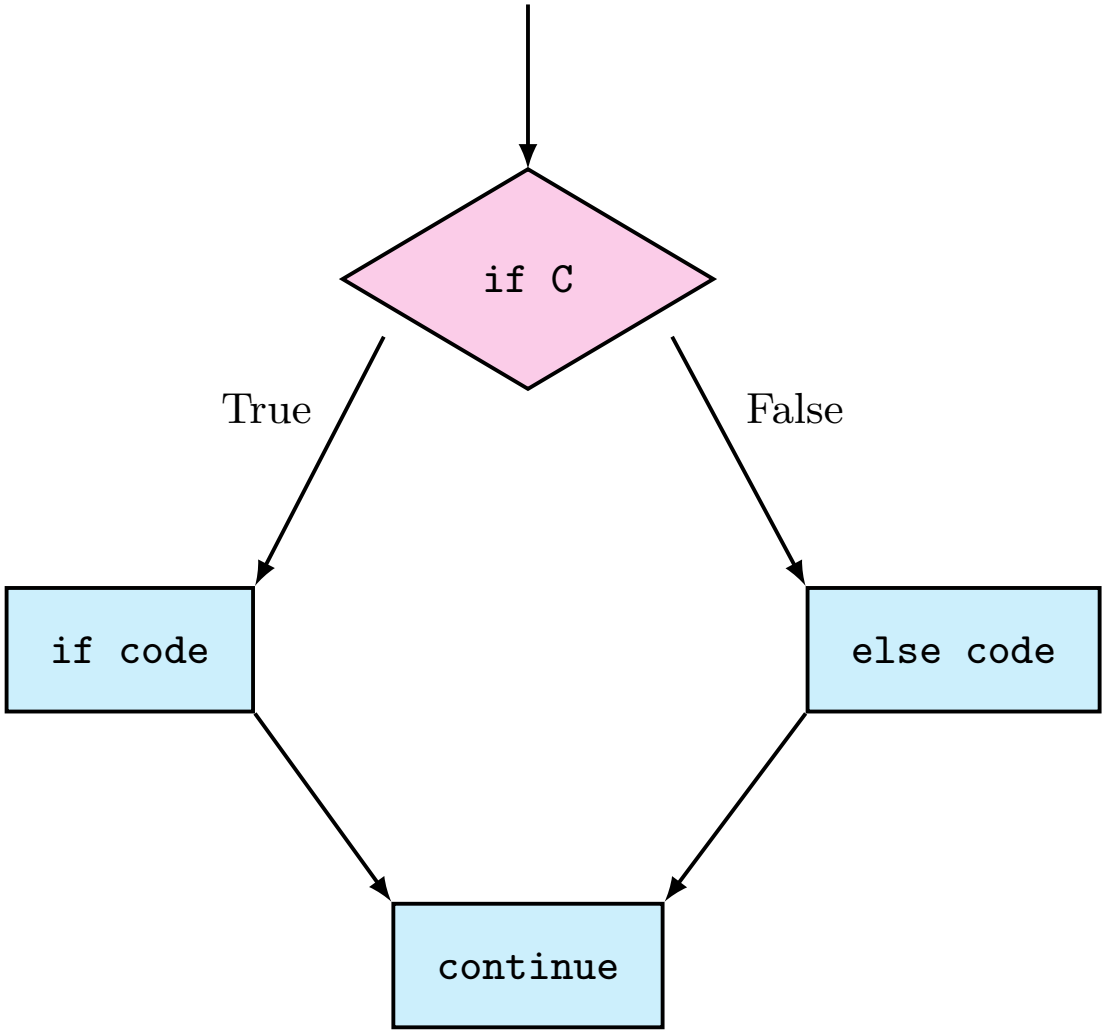

Figure 5: The if-then-else control structure. Unlike Figure 4 there is no bypass edge: every path through the diamond runs exactly one of the two blocks before the branches rejoin.

Elif chains

```
if <cond0>:
    <code>
elif <cond1>:
    <code>
...
elif <condN>:
    <code>
```

Each `elif` condition is only checked if every condition above it was `False` — so later branches can safely assume the earlier conditions failed. If *all* of them fail, nothing runs at all.

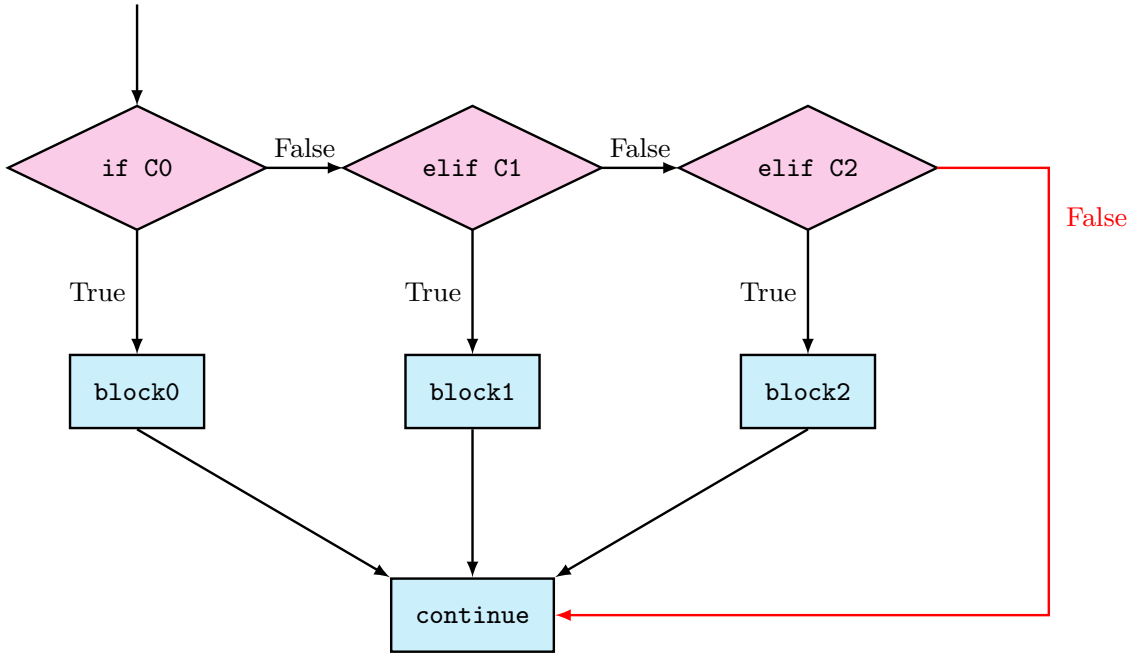

Figure 6: An if/elif/elif chain. Each False edge feeds sideways into the next test, so at most one block runs; the red edge is the path taken when every condition is False, and it runs no block at all. That red path is exactly why a variable first assigned inside such a chain can end up undefined.

Common bug: overlapping conditions in the wrong order

Because a later `elif` implicitly assumes the earlier ones were false, checking overlapping ranges from the *loosest* condition down silently picks the wrong branch:

```
>>> age = 60

>>> if age >= 18:
...     beverage = "cheap beer"
... elif age >= 30:
...     beverage = "standard beer"
... elif age >= 50:
...     beverage = "expensive beer"

>>> beverage
'cheap beer'
```

This code does not work as intended — see 2025-08-11-sequence-selection-and-iteration[22] for the traced flowchart of exactly where `age = 60` goes.

Fix 1 — bound each range explicitly. Once the ranges are disjoint the `elif`s are not doing any work; three plain `if`s would behave identically:

```
>>> age = 60

>>> if 18 <= age < 30:
...     beverage = "cheap beer"
... elif 30 <= age < 50:
...     beverage = "standard beer"
... elif 50 <= age:
...     beverage = "expensive beer"

>>> beverage
'expensive beer'
```

Fix 2 — order the tests from the *tightest* condition down, and let each `elif` inherit the guarantee that the ones above it failed:

```
>>> age = 60

>>> if age >= 50:
...     beverage = "expensive beer"
... elif age >= 30:          # guaranteed age < 50
...     beverage = "standard beer"
... elif age >= 18:          # guaranteed age < 50 and age < 30
```

[22] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf

```
...     beverage = "cheap"

>>> beverage
'expensive beer'
```

Still no catch-all

Fix 2 is correct for adults and silently broken for everyone else — with no branch taken, `beverage` is never assigned:

```
>>> age = 9

>>> if age >= 50:
...     beverage = "expensive beer"
... elif age >= 30:
...     beverage = "standard beer"
... elif age >= 18:
...     beverage = "cheap"

>>> beverage
NameError: name 'beverage' is not defined
```

If-elif-else

```
if <cond0>:
    <code>
elif <cond1>:
    <code>
...
elif <condN>:
    <code>
else:
    <code>
```

`else` is the catch-all: it runs whenever none of the preceding `if`/`elif` conditions were `True`, which closes off the `NameError` above.

```
>>> age = 9

>>> if age >= 50:
...     beverage = "expensive beer"
```

```
... elif age >= 30:
...     beverage = "standard beer"
... elif age >= 18:
...     beverage = "cheap"
... else:
...     beverage = "apple juice"

>>> beverage
'apple juice'
```

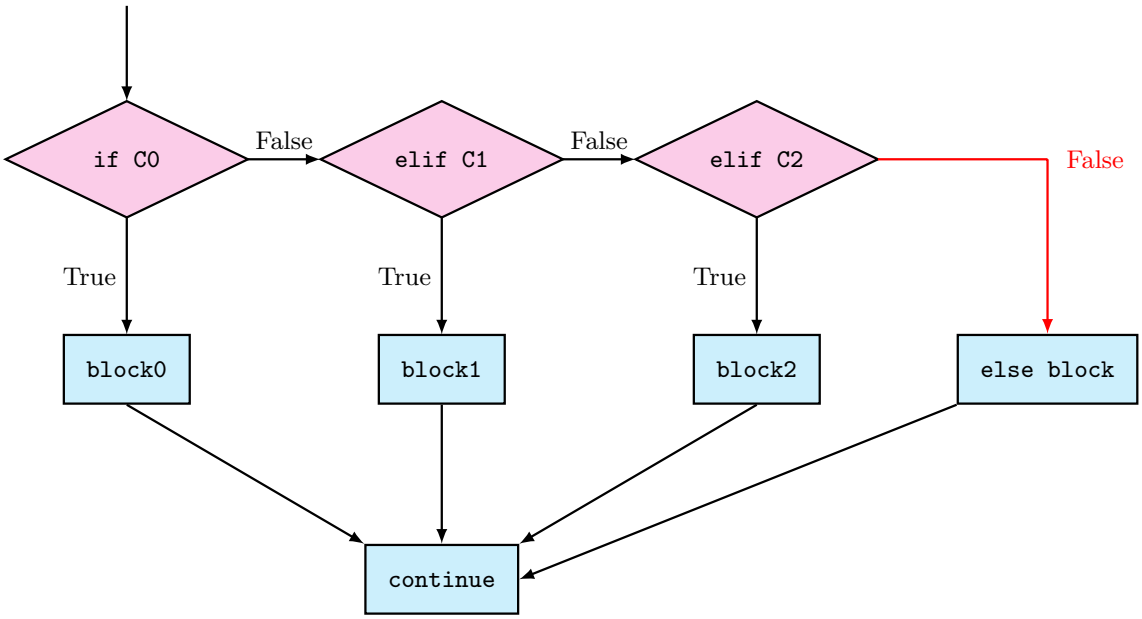

Figure 7: The same chain with an else. Compare with Figure 6: the red all-false edge no longer runs straight to the join, it runs into a block. That single structural difference is the whole of the NameError-versus-apple-juice contrast above.

Factoring and refactoring

Definition (factoring). In computer science **factoring** means breaking a complex problem into parts that are easier to conceive, understand, program, and maintain.

Definition (refactoring). Code **refactoring** is the process of restructuring existing computer code — changing the factoring — without changing its behaviour.

Simplifying if-statements

Nested `if`s with no other body collapse into a single `and`ed condition:

```
if x>1:                          if x>1 and y>2 and z>3:
    if y>2:              ==>         print("hello")
        if z>3:
            print("hello")
```

An if/else that only returns `True`/`False` can be replaced by the condition itself — the condition *is already* a boolean:

```
def foo(x):                      def foo(x):
    if x > 0:            ==>         return x > 0
        return True
    else:
        return False
```

The same for an if/else that only assigns:

```
if x > 0:                        y = x > 0
    y = True          ==>
else:
    y = False
```

And comparing a condition to `True`/`False` is redundant:

```
if x > y == True:     ==>    if x > y:
if x > y == False:    ==>    if not x > y:
```

`elif` can drop what the previous condition already ruled out

These two blocks **are** equivalent, because reaching the `elif` already guarantees `x <= 0`:

```
>>> if x > 0:                    >>> if x > 0:
...     print("A")                ...     print("A")
... elif x <= 0 and x % 2 == 0:  ... elif x % 2 == 0:
...     print("B")                ...     print("B")
```

These two are **not**, because the second `if` is independent of the one before it and does not know it already ran:

```
>>> if x > 0:                    >>> if x > 0:
...     print("A")                ...     print("A")
>>> if x <= 0 and x % 2 == 0:    >>> if x % 2 == 0:
...     print("B")                ...     print("B")
```

Let `x = 2`. The first block prints `A`; the second prints `A` **and** `B`.

Common errors

`or` does not distribute over bare values. The following will *always* print:

```
>>> if x == 1 or 2 or 3:
...     print("hello")
```

because it parses as `(x == 1) or 2 or 3`, and `2` is truthy on its own regardless of `x`. It is **not** equivalent to

```
>>> if x == 1 or x == 2 or x == 3:
...     print("hello")
```

which refactors to

```
>>> if x in [1,2,3]:
...     print("hello")
```

An empty branch is a wasted branch. This *is* a genuine equivalence rather than a bug, but the left-hand form is a roundabout way of writing the right-hand one:

```
>>> if x:                        >>> if not x:
...     pass                      ...     print("Hello World")
... else:
...     print("Hello World")
```

Python PEP8 Style Guide

PEP stands for **Python Enhancement Proposal**. A PEP is a design document providing information on how to **style** our code. This course follows **Google's PEP**, which is a variant of **PEP8**. Give it a browse.

Variable and function names

Names **must start with a letter** and must not include special characters excepting the underscore `_`.

Function and variable names should be lowercase, with words separated by underscores as necessary to improve readability:

Yes	No
`descriptive_variable_name`	`DescriptiveVariableName`

Spacing

Put single spaces around the *loosest-binding* operators in an expression and close up the tighter ones, so the spacing mirrors the precedence. A bracketed subexpression is treated as an atom whose interior is closed up.

Yes	No
`i = i + 1`	`i=i+1`
`x = x*2 - 1`	`x = x * 2 - 1`
`hypot2 = x*x + y*y`	`hypot2 = x * x + y * y`
`c = (a+b) * (a-b)`	`c = (a + b) * (a - b)`

Breaking long computation

According to PEP8 all lines must be **strictly less than 80 characters** wide.

```
>>> income = (gross_wages
...           + taxable_interest
...           + (dividends - qualified_dividends)
...           - ira_deduction
...           - student_loan_interest)
```

The **redundant bracket that encloses the entire expression** is what enables the expression to be broken across multiple lines — usually, hitting enter would evaluate the expression instead.

Comments

Anything following a `#` will be ignored by Python. In addition to docstrings (see python-functions[23]) you can — and *should* — include comments in your code to explain something not obvious in your design.

Not helpful:

```
x = x + 1                       # Increment x
```

Helpful:

```
x = x + 1                       # Compensate for border
```

The first comment restates what the code already says. The second says *why*.

Python Primitive Data Types

The types "built-in" to Python by default are called **primitive data**: booleans, integers, floats, strings, and tuples. All of them are **immutable** — once created, an object of one of these types cannot be changed.

Booleans

The boolean type provides, comprehensively, the values `True` and `False`:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

[23] courses/csse1001/python-functions.pdf

Comparison operators

Having a boolean type is what makes **comparison operators** useful — they return a true/false answer:

```
>>> 3 == 9
False
>>> 3 > 7
False
>>> 3 >= 7
False
```

All comparison operators have **equivalent precedence** and are **left associative**; when mixed with arithmetic, comparison is evaluated *last* (see python-operator-precedence-and-associativity[24]).

Operator	Description
`==`	Equal
`!=`	Not equal
`<`, `<=`	Less than, less than or equal
`>`, `>=`	Greater than, greater than or equal

```
>>> 3 + 2 > 1 + 3   # arithmetic first
True
>>> 3+2 > 1+3       # better to group like this
True
>>> 3 + (2>1) + 3   # True has int value 1
7
```

Logical connectors

Functions that return booleans are called **predicates**, and are (very) important to programming. More sophisticated predicates are built with the logical connectors `and`, `or`, and `not` — covered in detail in 2025-08-11-sequence-selection-and-iteration[25] and python-boolean-logic[26].

[24] courses/csse1001/python-operator-precedence-and-associativity.pdf
[25] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf
[26] courses/csse1001/python-boolean-logic.pdf

`bool` — truthy / falsy

Any object can be converted to a boolean with `bool`. Data that converts to `True` is **truthy**; data that converts to `False` is **falsy**. Usually the falsy element is whatever acts as zero in the type, and everything else is truthy.

```
>>> bool(0)
False
>>> bool(1)
True
```

Integers

> **Definition (Integer).** An **integer** is a number *without* a fractional part. It can be zero, positive, or negative.

Integers are **unbounded** in Python: we can work with integers of arbitrary size without worrying about **overflow**, which is not typical of programming languages.

```
>>> 2 ** 256 - 1
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

Any object can be converted to an integer with `int`. `True` behaves as `1` and `False` as `0`, so booleans can be used in arithmetic:

```
>>> int(True)
1
>>> int(False)
0
>>> True + False
1
>>> True * False
0
```

Strings

> **Definition (String).** Anything (with some exceptions) enclosed by single-quotes `' '` or double-quotes `" "` is considered a **string** by Python. A string is an **ordered collection** of the characters (e.g. unicode and ascii) allowed by the computer.

The quotes are what make it a string; without them the same text is either a syntax error or a name:

```
>>> "hello world"
'hello world'
>>> type("hello world")
<class 'str'>
>>> hello world   # note the lack of quotes
SyntaxError: invalid syntax
>>> hello         # note the lack of quotes
NameError: name 'hello' is not defined
```

`str` conversion and formatting

Any object can be converted to a string with `str`:

```
>>> str(1)
'1'
>>> str(True)
'True'
>>> str("Hello")
'Hello'
```

Substitution converts values into strings inside a sentence, using the string formatter — an `f` before the quotes, and `{}` around each expression to substitute:

```
>>> x = 1
>>> y = "two"
>>> z = f"x is {x} y is {y}"     # formatted string
>>> z
'x is 1 y is two'
>>> print(f"x is {x} y is {y}")
x is 1 y is two
>>> f"{x}"                       # alternative to str
'1'
```

Concatenation and scalar multiplication

Adding strings concatenates them — and because a string can be added to itself repeatedly, it also makes sense to multiply a string by a positive integer:

```
>>> "hello" + "world"
'helloworld'
>>> type(" ")
<class 'str'>
>>> space = " "
>>> "hello" + space + "world"
'hello world'
>>> "Hello World!" + "Hello World!" + "Hello World!"
'Hello World!Hello World!Hello World!'
>>> 3 * "Hello World!"
'Hello World!Hello World!Hello World!'
```

When strings are added a **new string is created** — the originals are untouched, which is what immutability forces.

Comparing strings

Strings compare by character order; `ord` gives a character's order and `chr` recovers the character from the order:

```
>>> "a" < "b"
True
>>> ord("a"), ord("b")   # the "order" of the characters
(97, 98)
>>> chr(97)              # get character from order
'a'
>>> "A" < "a"
True
>>> "Z" < "a"
True
```

Comparison then proceeds character-by-character, with a shorter string counting as less than any extension of itself:

```
>>> "a" < "aa"
True
>>> "b" < "aa"
False
>>> "aba" < "ab"
False
>>> "aZ" < "aa"
True
```

Exercise. Write `string_less_than(cs: str, ds: str) -> bool`, returning `True` only when `cs < ds`, restricting yourself to comparing integers only. `string_less_than("aba", "ab")` is false and `string_less_than("aZ", "aa")` is true.

Escape characters

A **new line** is an escape character that prints whatever follows it on a new line; the escape character is `\n`. A **tab** (`\t`) is a fixed amount of horizontal space, though how it is displayed depends on the program doing the displaying — which is why tabs are the worst.

```
>>> "hello\nworld"
'hello\nworld'
>>> print("hello\nworld")
hello
world
>>> "hello\tworld"
'hello\tworld'
>>> print("hello\tworld")
hello   world
```

A string can be **stored** differently than it is **printed**.

Numbers versus strings

`+` never silently converts between `int` and `str` — mixing them is a `TypeError`, and you have to say which conversion you meant:

```
>>> 3 + 7
10
>>> "3" + "7"
'37'
>>> 3 + "7"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> str(3) + "7"
'37'
>>> 3 + int("7")   # convert string to integer
10
```

Converting a string to a number only works for numbers expressed with digits, and `int` will not take a fractional literal:

```
>>> int("3.14")
ValueError: invalid literal for int() with base 10: '3.14'
>>> float("123.456")
123.456
>>> int("seven")
ValueError: invalid literal for int() with base 10: 'seven'
```

Length and inclusion

A string's **length** is the number of characters that comprise it:

```
>>> len("h")
1
>>> len("hello")
5
>>> cs = "world"
>>> len(cs)
5
>>> len(cs+"world") == len(cs) + len("world")
True
```

We can ask whether a string is a **substring** of another with `in`:

```
>>> "h" in "hello world"
True
>>> "hello" in "hello world"
True
>>> cs = "world"
>>> cs in "hello world"
True
>>> "ow" in "hello world"
False
```

Indexing

Because a string is *ordered*, we can number its characters **starting from zero** and access them with square brackets. We can also index from the end with negative numbers:

```
>>> cs = "hello world"
>>> cs[0]
'h'
```

```
>>> cs[1]
'e'
>>> cs[2]
'l'
>>> cs[len(cs)]
IndexError: string index out of range
>>> cs[-1]
'd'
>>> cs[-2]
'l'
>>> cs[-11]
'h'
```

Slicing

A slice `cs[start:stop:step]` takes *start* inclusive through *stop* exclusive, moving by *step*. The examples use `cs = "0123456789"` throughout, so that each index matches the character sitting at it.

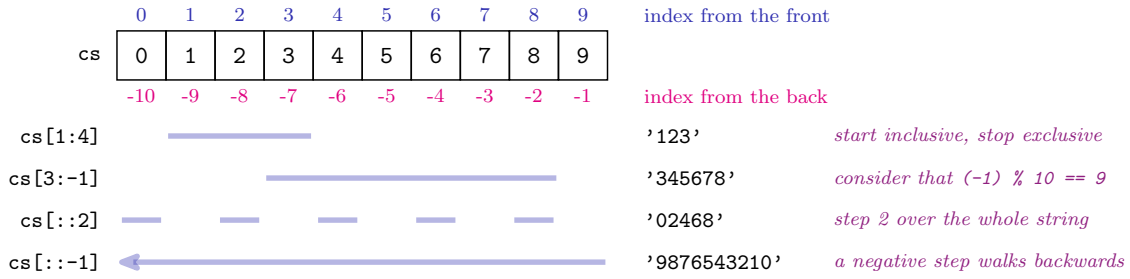

Figure 8: The string `cs = '0123456789'` with both index systems and four of its slices. Positive indices count from the front, negative from the back, and the two agree because a negative index i names position $i \bmod 10$ — which is why `cs[3:-1]` stops at 8, one before position 9.

Omitting an endpoint defaults it to the far end in the direction of travel:

```
>>> cs = "0123456789"
>>> cs[1:4]      # 1st inclusive through 4th exclusive
'123'
>>> cs[0:9]
'012345678'
>>> cs[0:10]
'0123456789'
```

```
>>> cs[0:]       # omitting the right endpoint defaults to the line above
'0123456789'
>>> cs[:-1]      # omitting the left endpoint defaults to zero
'012345678'
>>> cs[:]        # omitting both gives everything
'0123456789'
```

Negative endpoints are the same positions counted from the back — read a negative index modulo the length:

```
>>> cs[-1] == cs[len(cs)-1]
True
>>> cs[3:-1]     # consider that (-1) % 10 == 9
'345678'
>>> cs[-7:]      # consider that (-7) % 10 == 3
'3456789'
```

A step may be given third, and a negative step travels backwards:

```
>>> cs[::2]      # increment by two, zero inclusive to ten exclusive
'02468'
>>> cs[0:10:2]   # the above defaulted to this
'02468'
>>> cs[::-1]     # decrement by one, ten inclusive to zero exclusive
'9876543210'
>>> cs[10::-1]   # the above defaulted to this
'9876543210'
>>> cs[:10:-1]   # not to this, because we are moving backwards
''
```

Mixing all three, forwards and backwards:

```
>>> cs[1:-3:3]    # increment by three, one inclusive to seven exclusive
'14'
>>> cs[-8:-2:2]   # increment by two, two inclusive to eight exclusive
'246'
>>> cs[-8:-2:-4]  # decrement by four, two inclusive to eight exclusive
''
>>> cs[::-4]      # decrement by four, ten inclusive to zero exclusive
'951'
```

Immutability

Something is **immutable** when it cannot be changed. **Strings are immutable** — so a word cannot be capitalised by writing over its first character:

```
>>> cs = "hello"
>>> cs[0] = "H"
TypeError: 'str' object does not support item assignment
```

Strings as booleans

The empty string is the falsy string; every other string is truthy. It is also "smaller" than every other character:

```
>>> bool("a")
True
>>> bool("Hello")
True
>>> bool("")
False
>>> "" < "A"
True
>>> "" < chr(0)
True
```

A space is not the same thing as empty:

```
>>> space = " "      # a space
>>> len(space)
1
>>> bool(space)
True
>>> empty = ""       # empty string
>>> len(empty)
0
>>> bool(empty)
False
```

Tuples

> **Definition (tuple).** A **tuple** is an *immutable* ordered collection of elements. These elements are *not necessarily* the same type and are *not* necessarily distinct. Round brackets `()` are used to create tuples in Python.

Being ordered, tuples index and slice exactly like strings; being immutable, they refuse item assignment exactly like strings:

```
>>> xs = (0, 1, 2, 3, 4, 5)
>>> type(xs)
<class 'tuple'>
>>> xs[-1]        # indexing
5
>>> xs[0:4]       # slicing
(0, 1, 2, 3)
>>> xs[0] = -1    # immutable
TypeError: 'tuple' object does not support item assignment
```

Tuples support the same addition and scalar multiplication as strings, and their elements can be of mixed type:

```
>>> xs = (1, 'a', 2, 'b')    # mixed types are fine
>>> ys = (3, 'c', 4, 'd')
>>> xs + ys                  # tuple addition
(1, 'a', 2, 'b', 3, 'c', 4, 'd')
>>> 2*xs                     # tuple scalar multiplication
(1, 'a', 2, 'b', 1, 'a', 2, 'b')
```

Nomenclature

Tuples are named by size: couple (2), triple (3), quadruple (4), quintuple (5), sextuple (6), septuple (7), octuple (8), and an n-tuple in general. The general form `(0, 1, 2, ..., n-1)` is *not* valid Python syntax — it is informal notation for the pattern.

Singleton and empty tuples

A single value in round brackets is **not** a tuple — brackets there are just grouping. The trailing comma is what makes a singleton:

```
>>> (1) + (2)
3
>>> type((1))
<class 'int'>
>>> (1)
1
>>> type((1,))
<class 'tuple'>
>>> (1,) + (2,)
(1, 2)
>>> (1) + (2,3)   # common error
TypeError: unsupported operand type(s)
```

The empty tuple is `()` — a genuine tuple, the identity for tuple addition, and falsy:

```
>>> ()
()
>>> type(())
<class 'tuple'>
>>> () + (1,)
(1,)
>>> (1,) + ()
(1,)
>>> bool(())
False
```

Comparing tuples

Tuples compare element-by-element, and — like strings — a shorter tuple is less than a longer tuple that extends it:

```
>>> (1, 2, 3) == (1, 2, 3)
True
>>> (1, 2, 3) == (1, 2)
False
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3) < (1, 2)
False
>>> (1, 2) < (1, 2, 3)
True
```

Packing and unpacking

Several names can be assigned on one line, and several values printed on one line, through an (implicit) tuple:

```
>>> x, y, z = 2, 3, 4    # saves vertical space
>>> x
2
>>> y
3
>>> z
4
>>> x, y, z              # repl-printing several values
2, 3, 4
```

A real REPL echoes the last line as the tuple it built, brackets included — `(2, 3, 4)` — which is the more useful way to read it: the right-hand side of a packing assignment is a tuple.

Python Testing (doctest & Assertions)

This is the reference note for testing in CSSE1001. 2025-09-01-testing[27] (Lecture 6C) develops it end to end. Two things it leans on come from 2025-09-01-string-methods[28] (Lecture 6A) and 2025-09-01-file-io[29] (Lecture 6B): the **REPL transcript**, which is the form a doctest is written in, and the **file pointer**, which is what makes file-reading functions awkward to test. Both are drawn below. Drawing conventions follow csse1001-diagram-notation[30].

`doctest`

A **docstring test** is an `>>>` example embedded in a function's docstring. `doctest.testmod()` actually *runs* every such example in the current module and reports failures:

```
>>> import doctest
>>> doctest.testmod()
TestResults(failed=0, attempted=2)
```

[27] courses/csse1001/2025-09-01-testing.pdf
[28] courses/csse1001/2025-09-01-string-methods.pdf
[29] courses/csse1001/2025-09-01-file-io.pdf
[30] courses/csse1001/csse1001-diagram-notation.pdf

`doctest.testfile(path)` instead runs every `>>>` example found in an arbitrary text file (useful for a test suite kept *outside* the functions being tested).

Nothing runs a doctest by itself — a docstring full of examples is inert documentation until `testmod` or `testfile` is called on it.

A doctest is a REPL transcript

A REPL transcript is exactly the shape a doctest takes:

- `>>>` marks a line of **input**;
- `...` marks a **continuation** line, so an entire `def` can be typed this way;
- the unprompted line(s) that follow are the **expected output**;
- a blank line, or the end of the docstring, ends the example.

So a literal transcript can be pasted into a docstring as it stands and becomes a test. The exceptions are transcripts that are abbreviated rather than literal (the `strip` example below) or that mark newlines with a pilcrow (¶). This `title` example is already a complete doctest:

```
>>> "a tale of two cities".title()
'A Tale Of Two Cities'
```

as are the `find` transcript below and the `split/join` transcripts in 2025-09-01-string-methods[31].

What a doctest actually compares

`doctest` compares Python's *exact printed output* against the expected text — not whether two values are `==`. The rule underneath that is a rule about the REPL, and it has two halves.

The blank line `print` leaves behind cannot be written literally as expected output, because a blank line is how a doctest example *ends* (see the anatomy above). `doctest` spells it `<BLANKLINE>` instead. It is one more reason to prefer the echoing form.

Two practical consequences of the echo rule:

- **The quote marks are part of the expected text.** `"hello".capitalize()` is expected to give `'Hello'`, and the `.title()` example above `'A Tale Of Two Cities'` — with the quotes. A doctest expecting bare `Hello` fails.
- **A call that returns `None` echoes nothing at all.** `print` is such a call: no `None` appears after the printed lines. So an example that only calls `print` has no expected-output line for the return value.

[31] courses/csse1001/2025-09-01-string-methods.pdf

```
>>> line = file.readline()
>>> line                                   an expression: the REPL echoes repr(line)
'What a\n'                                 the quotes and the \n are part of the text
```

```
>>> print(line)                            a statement: print writes str(line), then adds a newline
What a
¶                                          the ¶ is the blank line print leaves behind
```

expression → value → `repr(value)` → text doctest compares

`print(x)` → `str(x)` + newline → text doctest compares

Boxed: the exact text a doctest must reproduce. One value, two texts — so two different doctests.

Figure 9: What text a doctest has to reproduce. Evaluating the expression `line` echoes its repr, `'What a\n'`, quote marks and newline escape included. `print(line)` writes the string itself and adds a newline, which leaves a blank line; the magenta pilcrow on a line of its own marks it. Each magenta box holds the exact text doctest matches, and the two arrow chains give the rule for each case.

More pitfalls, all instances of the same rule:

- Whitespace matters — `[ ]` != `[]`, and Python always prints `, ` (comma-space) inside collection literals.
- Unordered types (`set`, `dict`) don't have a guaranteed print order for non-numeric elements — compare with `==` inside the doctest instead of relying on printed order:

  ```
  >>> {3, 1, 2} == some_function({1, 2, 3})
  True
  ```

- Floats are inexact — compare with a tolerance instead of exact equality:

  ```
  >>> abs(f(x) - expected) < 10**-3
  True
  ```

Why `repr` is the friendlier half for testing

Because the echo goes through `repr`, whitespace that is invisible when printed is visible in a doctest. That is what makes `str.strip()` testable at all: `strip` returns a copy with leading and trailing whitespace removed, and a newline counts as whitespace. Abbreviated, the example is the literal `"  123  \n"` on the input line and `123` on the output line, which drops both the `.strip()` call and the quotes. Written out as a doctest it is:

```
>>> "   123   \n".strip()
'123'
```

and there the difference between `'   123   \n'` and `'123'` is plainly on the page. Printed, the two are nearly indistinguishable.

Writing a comprehensive doctest

1. Typical cases *and* edge cases.
2. The zero of the data type (`0`, `[]`, `""`).
3. The singleton of the data type (`1`, `[1]`, `"a"`).
4. Correctness, not contract violations (don't test precondition failures).
5. No redundant tests.

Worked example: centring a word

The exercise: given a word and an integer width, centre the word in a sequence of `x`s. The answer is `str.center`, whose help text is

```
center(self, width, fillchar=' ', /)
    Return a centered string of length width.
    Padding is done using the specified fill character
    (default is a space).
```

The calls are `'spam'.center(10, 'x')` and `'101'.center(20, 'x')`. Three transcripts of typical calls *are* three doctest examples already; the checklist above turns them into a suite.

The three typical examples might look redundant against checklist rule 5. They are not: `'101'` in width 20 leaves an *odd* amount of padding to share out, while `'spam'` in 10 and `'UQQU'` in 30 leave *even* amounts, and `str.center` gives the leftover character to the right-hand side. That is two different paths through the method, not one tested three times.

> **i** An oracle for re-implementations
>
> Implement any string method you use at least once. When you do, the built-in method is a ready-made oracle: a doctest can assert your version agrees with it, e.g. `>>> my_center('spam', 10) == 'spam'.center(10, 'x')` expecting `True`.

```
def center_word(word:  str, width:  int) -> str:
    """ Centre word in a field of x's of the given width.
    >>> center_word('spam', 10)                                   typical
    'xxxspamxxx'
    >>> center_word('101', 20)          typical, odd padding: 8 on the left, 9 on the right
    'xxxxxxxx101xxxxxxxxx'
    >>> center_word('UQQU', 30)                  typical, even padding: 13 each side
    'xxxxxxxxxxxxxUQQUxxxxxxxxxxxxx'
    >>> center_word('', 4)                                        the zero of str
    'xxxx'
    >>> center_word('a', 1)                                     the singleton of str
    'a'
    >>> center_word('spam', 2)                  edge: no room to pad, so nothing is padded
    'spam'
    """
    return word.center(width, 'x')
```

The three examples beside the magenta bar come from the coverage checklist.
Their outputs were checked against CPython.

Figure 10: The centring exercise written as a doctest, with each example labelled with the job it does. The function is `center_word`, and its body is `str.center` with `'x'` as the fill character. The first three examples are typical cases. The three beside the magenta bar come from the coverage checklist above: the zero, the singleton and an edge case. The whole docstring passes testmod with six attempted, zero failed.

Optional parameters multiply the calling forms

The help text for `find` is

```
S.find(sub[, start[, end]]) -> int
```

The square brackets mean *optional*, and the rule **recurses** — an optional parameter can itself carry optional parameters. So the legal calling forms are exactly

```
"team".find("I")        "team".find("I", 1)        "team".find("I", 1, -1)
```

and `find(1, -1)` is illegal because `sub` is required. Each form is a different path into the function, so a comprehensive suite needs one example per form. This transcript covers three of them plus the failure return:

```
>>> "hello world".find("world", 6)
6
>>> "hello hello".find("hello")
0
>>> "hello hello".find("hello", 1)
6
>>> "hello hello".find("hello", 1, 3)
-1
```

The last line is worth its place on its own: the help text says *return -1 on failure*, so `-1` is part of the contract, not an error condition, and a suite that never provokes it has not tested the documented behaviour.

Testing code that reads a file

The same function can be written with two different signatures, and the difference matters for testing.

```
def num_empty_lines(path: str) -> int:      # takes a file name
    ans = 0
    with open(path, 'r') as file:
        for line in file:
            if line == '\n':
                ans = ans + 1
    return ans
```

```
from typing import TextIO                    # takes a file pointer

def num_empty_lines(file_pointer: TextIO) -> int:
    ans = 0
    for line in file_pointer:
        if line == '\n':
            ans = ans + 1
    return ans
```

These are different functions: the first takes a file **name**, the second a **pointer** to a file. The testing consequence follows from how the pointer behaves.

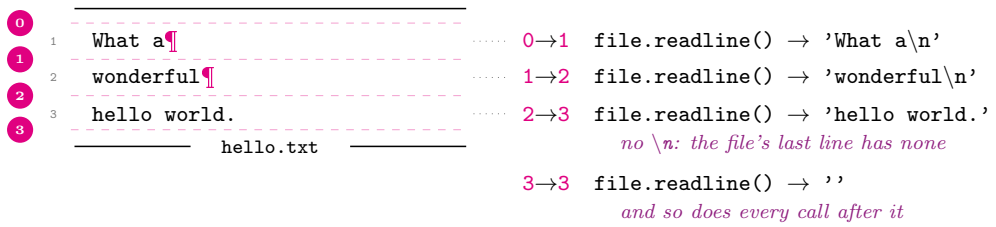

Figure 11: The file pointer walking hello.txt. A magenta pilcrow marks each newline in the file. There is no pilcrow after the last line, so that line comes back without a newline. The numbered circles are the pointer positions and the dashed lines are the line boundaries. Each `readline()` returns one line and moves the pointer past it. Once the pointer is at the end of the file, every further call returns the empty string.

The file-pointer version is not re-runnable. Doctest executes the examples in one docstring in order, in a single namespace, so a second call on the same handle reads an already-exhausted file:

```
>>> fp = open("notes.txt")      # a file with two empty lines
>>> num_empty_lines(fp)
2
>>> num_empty_lines(fp)
0
```

Both lines pass — and the second is a *lie* about the function. Either re-open the file for each example, or test the name-taking version, which opens the file itself and so starts from position **0** every time.

Either way, a file-reading doctest depends on the file system. A bare name is resolved relative to the directory Python is running in (see `os.getcwd()` and `os.chdir()`). So the

test passes or fails depending on the working directory, and it needs a fixture file to exist. `doctest.testfile` is one way out: keep the suite in a text file that sits beside the fixture it reads, so the two travel together.

> ⚠ Never open a fixture in `"w"` mode from a test
>
> A `"w"` open on `numbers.dat` creates the `numbers.dat` file, or **overwrites the old one** if it exists. A doctest that opens its own fixture with `"w"` destroys the fixture on the first run and then passes forever on an empty file. Use `"r"` to read it and `"a"` (which appends, and creates the file only if it is absent) if a test genuinely has to add to one.

Also prefer `with` in anything a test calls. An unclosed file stays open and vulnerable to side effects: some of the file may go missing, or extra bits may appear in it. `with` closes the file even if the program crashes while it is executing its code block, which is exactly what a failing test does.

Type hints document; only tests and assertions check

The two file-reading signatures above are a good place to see this. `TextIO` is the type hint for a file pointer, as in `num_empty_lines(file_pointer: TextIO)`. Write that parameter as `file_pointer: str` and the function runs just the same. The highest-rated-band function in the next section can annotate its parameter as `path: TextIO` and then call `open(path, 'r')` on it, and that runs too, although the parameter is really a `str` path. Both annotations disagree with their own code and Python runs them anyway. Nothing in the language checks an annotation. In this material the only things that check anything at run time are the doctests you write and the assertions you place.

Assertions

`assert <condition>` raises `AssertionError` (halting the program) if `<condition>` is false — useful for catching impossible situations as soon as they occur, rather than letting them silently propagate:

```
assert ans > 0   # all factorials are positive/non-zero
```

`assert False` documents a line that should be unreachable (e.g. after an exhaustive `if`/`else`).

An assertion is the executable form of a claim you would otherwise only argue in a comment. A function that finds the highest-rated band in a file is built on exactly such a claim: it initialises

```
current_highest_rating = -float('inf')      # Guarantees line 11 satisfied
```

where line 11 is the comparison `if rating > current_highest_rating:`. The comment is a piece of reasoning about program state — *this starting value is below every rating the file can hold, so the first row always wins* — and nothing checks it.

The same function shows what an assertion buys you. `current_most_popular_band` starts as `""`, so on a `bands.txt` that has only its header row the loop never runs and the function silently returns the empty string. Adding

```
assert current_most_popular_band != ""      # the file held at least one band
```

before the `return` turns that silent wrong answer into an `AssertionError` at the point it occurs.

See python-exceptions[32] for `AssertionError` among Python's other exceptions, and python-file-io[33] and python-string-methods[34] for the machinery the file and string examples above use.

[32] courses/csse1001/python-exceptions.pdf
[33] courses/csse1001/python-file-io.pdf
[34] courses/csse1001/python-string-methods.pdf