Sequence, Selection, and Iteration

Table of contents

Outline 1
Learning objectives 1
Part 1 — logic 2
Part 2 — the `if` statement 3
Part 3 — factoring and refactoring 5
Summary 6
Next lecture 6

Lecture 3B. Course logistics are in csse1001[1].

Outline

Despite the title, this is entirely about **selection** — *controlling program flow*. Sequencing is in 2025-08-04-python-memory-model[2], 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.

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

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:

- **`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[3].

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`.

[3] 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[4].

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'
```

[4] 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:

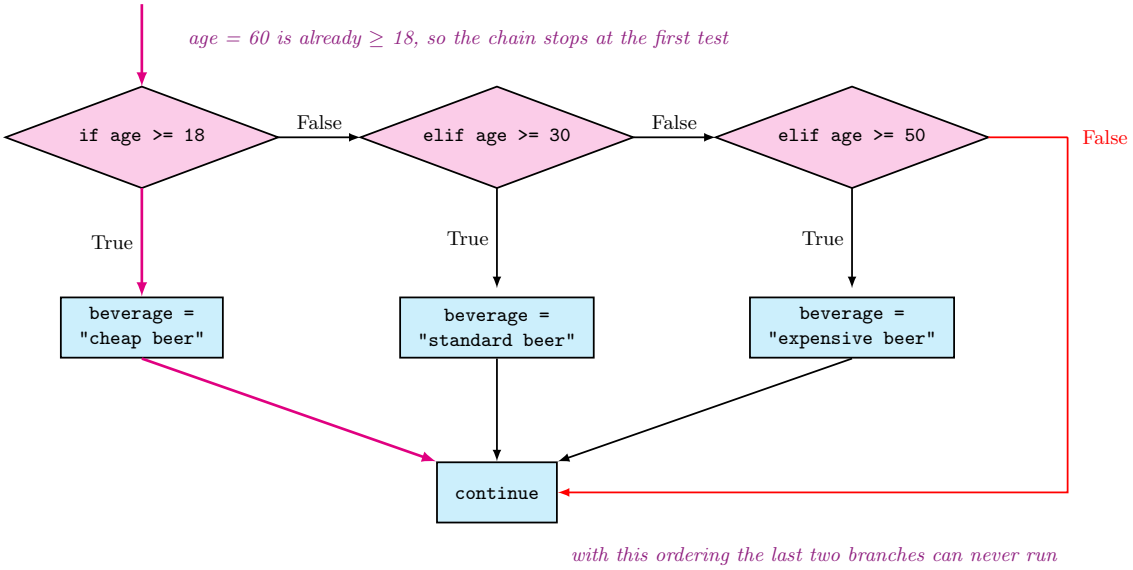

Figure 1: The buggy beverage chain as an if/elif/elif flowchart. The magenta trace is the path age = 60 takes. The red edge is the False exit from the last test, where no branch runs. Because age >= 18 is the loosest test and sits first, any age reaching the second diamond is already under 18 — so the 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[5].

[5] 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[6].

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.

[6] 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[7] — iteration.

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