Python If Statements

Table of contents

If-then 1
If-then-else 4
Elif chains 5
If-elif-else 9
Factoring and refactoring 11
Common errors 12

Definition (if-statement). Given a condition C — i.e. a predicate that evaluates to `True` or `False`, see python-boolean-logic[1] — 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>
    ...
```

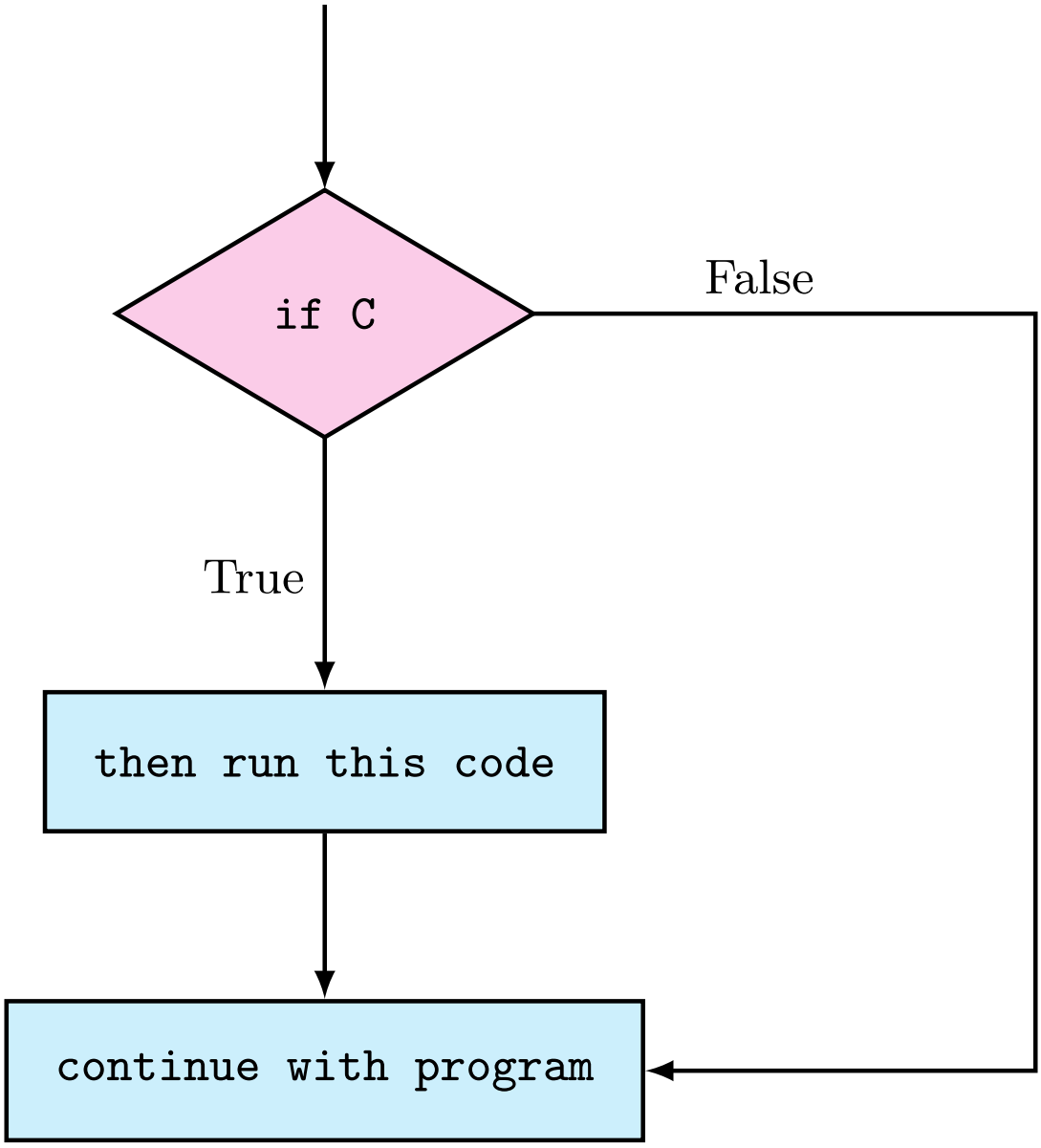

Figure 1: The if-then control structure. The True branch drops into the guarded block; the False branch bypasses it entirely and rejoins the program at the same place, which is why an if-statement can only ever *skip* code, never choose between alternatives.

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

Because of truthiness[2], 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 = x + 1
>>> x
1
```

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

[1] courses/csse1001/python-boolean-logic.pdf
[2] courses/csse1001/python-boolean-logic.pdf

```
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[3]) — the second `if` is never even reached:

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

>>> 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>
```

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

```
>>> 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`.

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.

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'
```

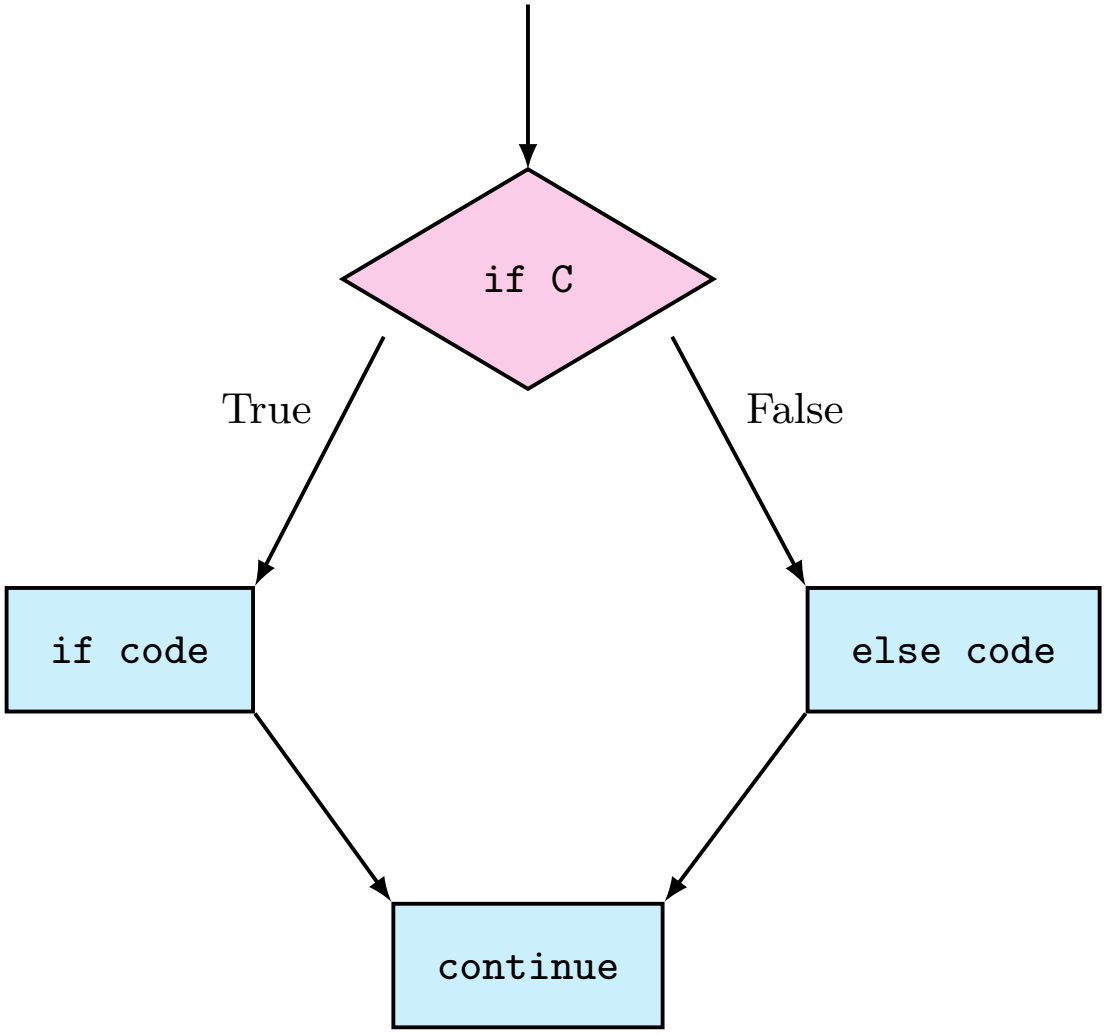

Figure 2: The if-then-else control structure. Unlike Figure 1 there is no bypass edge: every path through the diamond runs exactly one of the two blocks before the branches rejoin.

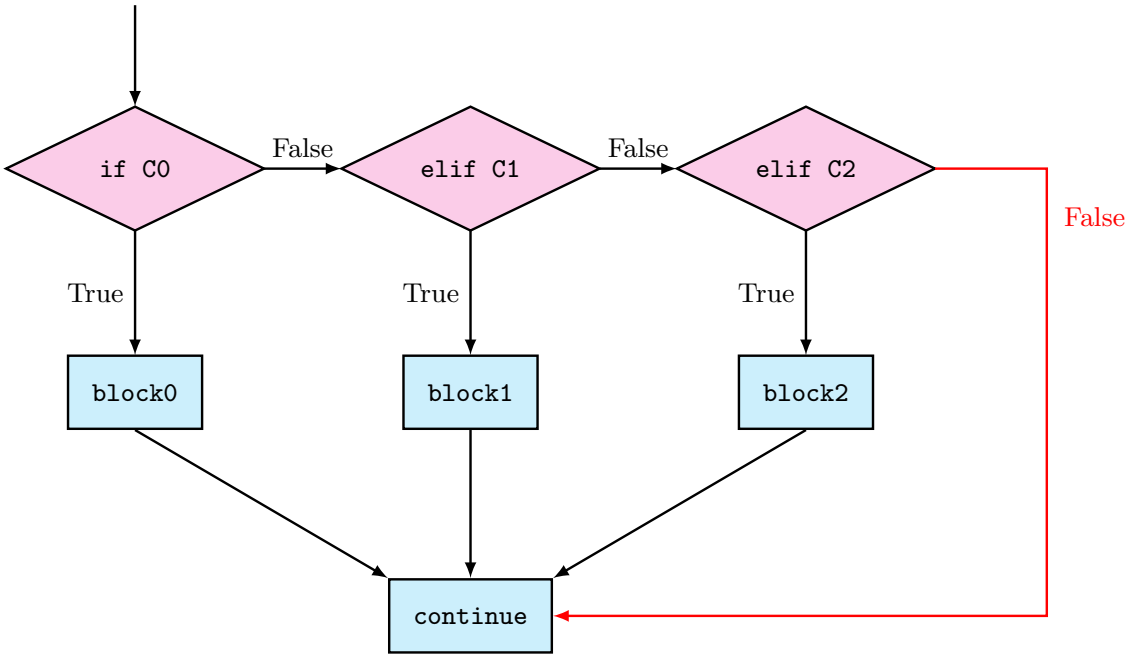

Figure 3: An if/elif/elif chain. Each False edge feeds sideways into the next test, so at most one block runs; the red edge is the path taken when every condition is False, and it runs no block at all. That red path is exactly why a variable first assigned inside such a chain can end up undefined.

This code does not work as intended — see 2025-08-11-sequence-selection-and-iteration[4] 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
...     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:

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

```
>>> 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'
```

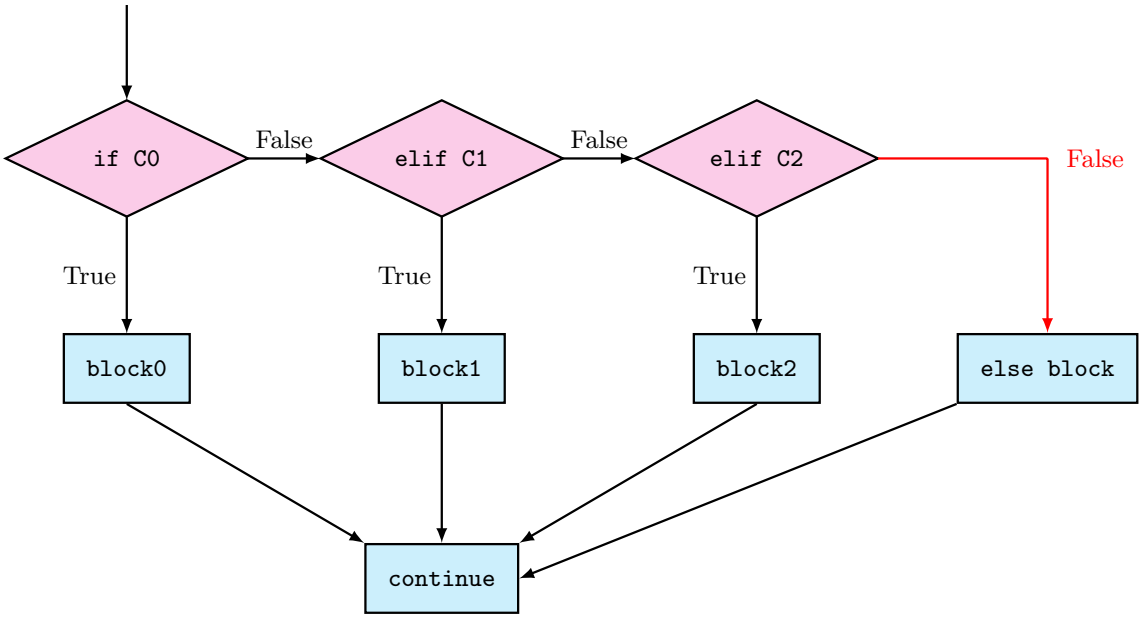

Figure 4: The same chain with an else. Compare with Figure 3: the red all-false edge no longer runs straight to the join, it runs into a block. That single structural difference is the whole of the NameError-versus-apple-juice contrast above.

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:
...     pass
... else:
...     print("Hello World")
```

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