Python If Statements

Table of contents

Given a condition C (a predicate — 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.

If-then

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

Only the indented code runs when the condition holds; execution otherwise skips straight past it. Because of truthiness[2], the condition doesn't need to literally be `True`/`False`:

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

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

```
>>> x
2
```

Warning. A variable only assigned inside an if-block does not exist if the condition was false:

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

If-then-else

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

`if-else` picks between exactly one of two instruction sets — unlike two separate `if`s, the condition is only checked once and the two branches can never both run.

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.

Common bug. Because later `elif`s implicitly assume the earlier ones were false, checking overlapping ranges in the wrong order 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'   # not what was intended!
```

Two ways to fix it — bound each range explicitly:

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

or check from highest to lowest, relying on the guarantee each `elif` gives about ranges already ruled out:

```
>>> 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 beer"
```

If-elif-else

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

`else` is a catch-all — it runs whenever none of the preceding `if`/`elif` conditions were `True` (avoiding a `NameError` from a variable never getting assigned).

Factoring and refactoring

Factoring means breaking a complex problem into parts that are easier to conceive, understand, program, and maintain. **Refactoring** is the process of restructuring existing code — changing the factoring — without changing its behaviour.

Simplifying if-statements

Nested `if`s can often 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 assigns/returns `True`/`False` can be replaced by the condition itself:

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

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

Comparing directly to `True`/`False` is redundant:

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

Equivalence of `elif` vs. nested `if`. An `elif` condition can safely assume the earlier condition was false, so `elif x <= 0 and x % 2 == 0` is equivalent to just `elif x % 2 == 0` (given a preceding `if x > 0`). This is *not* true for a second, independent `if` — `if x <= 0 and x % 2 == 0` is *not* equivalent to a bare `if x % 2 == 0` placed after `if x > 0`, since the second `if` doesn't know the first one already ran (e.g. with `x = 2`, the `elif` form prints only `A`, but the two-independent-`if`s form prints both `A` and `B`).

Common errors

`or`/`and` do not distribute over a list of bare values — `x == 1 or 2 or 3` always evaluates truthy (`2` and `3` are truthy on their own, regardless of `x`). The comparison needs to be repeated instead: `x == 1 or x == 2 or x == 3` (later refactored to `x in [1, 2, 3]`).

`if x: pass else: <code>` is just a roundabout way of writing `if not x: <code>`.