Python Boolean Logic

Table of contents

Boolean domain and predicates 1
`and` and `or` 2
Precedence 3
Short circuits (lazy evaluation) 4
Comparisons are still expressions 4
Truthiness 5
What `and`/`or` actually return 6

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[1]. 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
```

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:

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

```
>>> 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`.