Theory: Boolean logic

23 minutes 8 / 8 problems solved

§1. Boolean type

The **Boolean** type, or simply `bool`, is a special data type that has only two possible values: `True` and `False`. In Python the names of boolean values start with a capital letter.

In programming languages the boolean, or logical, type is a common way to represent something that has only two opposite states like *on* or *off*, *yes* or *no*, etc.

If you are writing an application that keeps track of door openings, you'll find it natural to use `bool` to store the current door state.

```
is_open = True
is_closed = False

print(is_open)    # True
print(is_closed)  # False
```

§2. Boolean operations

There are three built-in boolean operators in Python: `and`, `or` and `not`. The first two are **binary** operators which means that they expect two arguments. `not` is a **unary** operator, it is always applied to a single operand. First, let's look at these operators applied to the boolean values.

- **and** is a binary operator, it takes two arguments and returns `True` if both arguments are true, otherwise, it returns `False`.

```
a = True and True    # True
b = True and False   # False
c = False and False  # False
d = False and True   # False
```

- **or** is a binary operator, it returns `True` if at least one argument is true, otherwise, it returns `False`.

```
a = True or True    # True
b = True or False   # True
c = False or False  # False
d = False or True   # True
```

- **not** is a unary operator, it reverses the boolean value of its argument.

```
to_be = True           # to_be is True
not_to_be = not to_be  # not_to_be is False
```

§3. The precedence of boolean operations

Logical operators have a different priority and it affects the order of evaluation. Here are the operators in order of their priorities: `not`, `and`, `or`. So, `not` is considered first, then `and`, finally `or`. Having this in mind, consider the following expression:

```
print(False or not False)  # True
```

First, the part `not False` gets evaluated, and after evaluation, we are left with `False or True`. This results in `True`, if you recall the previous section.

While dealing solely with the boolean values may seem obvious, the precedence of logical operations will be quite important to remember when you start working with so-called **truthy** and **falsy** values.

Current topic:
✓ Boolean logic Stage 1 16★

Topic depends on:
✓ Variables Stage 1 17★

Topic is required for:
✓ Comparisons Stage 1 16★
✓ Operations with list Stage 3 6★
Launching web server Stage 1

Table of contents:
↑ Boolean logic

§4. Truthy and falsy values

Though Python has the boolean data type, we often want to use non-boolean values in a logical context. And Python lets you test almost any object for truthfulness. When used with logical operators, values of non-boolean types, such as integers or strings, are called **truthy** or **falsy**. It depends on whether they are interpreted as `True` or `False`.

The following values are evaluated to `False` in Python:

- constants defined to be false: `None` and `False`,
- zero of any numeric type: `0`, `0.0`,
- empty sequences and containers: `""`, `[]`, `{}`.

Anything else generally evaluates to `True`. Here is a couple of examples:

```
print(0.0 or False)  # False
print("True" and True)  # True
print("" or False)  # False
```

Generally speaking, `and` and `or` could take any arguments that can be tested for a boolean value.

Now we can demonstrate more clearly the difference in operator precedence:

```
# `and` has a higher priority than `or`
truthy_integer = False or 5 and 100  # 100
```

Again, let's break the above expression into parts. Since the operator `and` has a higher priority than `or`, we should look at the `5 and 100` part. Both `100` and `5` happen to be truthy values, so this operation will return `100`. You have never seen this before, so it's natural to wonder why we have a number instead of the `True` value here. We'll cover this surprising fact shortly. Coming back to the original expression, you can see that the last part `False or 100` does exactly the same thing, returns `100` instead of `True`.

> The operators `or` and `and` return one of their operands, not necessarily of the boolean type. Nonetheless, `not` always returns a boolean value.

Another tricky example is below:

```
tricky = not (False or '')  # True
```

A pair of parentheses is a way to specify the order in which the operations are performed. Thus, we evaluate this part of the expression first: `False or ''`. This operand `''` evaluates to `False` and `or` returns this empty string. Since the result of the enclosed expression is negated, we get `True` in the end: `not ''` is the same as `True`. Why didn't we get, say, a non-empty string? The `not` operator creates a new value, which by default has the boolean type. So, as stated earlier, the unary operator always returns a logical value.

§5. Short-circuit evaluation

The last thing to mention is that logical operators in Python are **short-circuited**. That's why they are also called **lazy**. That means that the second operand in such an expression is evaluated only if the first one is not sufficient to evaluate the whole expression.

- `x and y` returns **x** if **x** is falsy; otherwise, it evaluates and returns **y**.
- `x or y` returns **x** if **x** is truthy; otherwise, it evaluates and returns **y**.

For instance:

```
# division is never evaluated, because the first argument is True
lazy_or = True or (1 / 0)  # True

# division is never evaluated, because the first argument is False
lazy_and = False and (1 / 0)  # False
```

Those were the very basics of boolean values and logical operations in Python. It's definitely good to know them right from the beginning!

Report a typo

😐 Feedback sent!

Start practicing

Comments (74) Hints (1) Useful links (4) Show discussion