# Theory: Conversion to boolean





You already know that all objects in Python can be interpreted as boolean values. Objects that are evaluated to `True` are called **truthy**, objects that are evaluated to `False` are called **falsy**. The following values are falsy:

- some constants: `None` and `False`,
- zero: `0`, `0.0`, `0j`,
- empty containers such as a string `""`, a list `[]`, and others.

All other objects are evaluated to `True`.

This allows us to use objects of any type in boolean expressions. In this topic, we are going to learn when it can be useful and when you should explicitly convert objects into boolean values.

## §1. Truthy test

Since there are few objects in Python that are evaluated to `False`, there are not many cases when non-boolean values are used in logical expressions. The most common one is checking whether the given container is empty or not. Let's write a function that prints a list if it is not empty and the string `"empty list"` if it is the opposite.

```
def print_list(lst):
    if lst:
        print(lst)
    else:
        print('empty list')


print_list([2, 3, 4])  # [2, 3, 4]
print_list([])         # empty list
```

According to PEP8, writing `if lst` instead of `if len(lst) > 0` is preferable, but you should understand well which objects could be passed to your function. In this example, if the passed argument `lst` turns out to be `None`, this function prints `"empty list"`.

> Here and later in this topic, all examples use lists but this all can be applied to other containers in the same way.

There is a more compact way to implement the same function but it requires a deep understanding of how the `and` and `or` operators work with non-boolean values.

## §2. Logical operations with non-boolean values

You already know that given two boolean values, `and` returns `True` if both operands equal `True` while `or` returns `True` if at least one operand equals `True`. When the operands are of the arbitrary type, Python can apply `and` and `or` operators to them, but the result will be one of the operands rather than the boolean values `True` or `False`.

The tables below show what `and`, `or` and `not` operators return depending on whether their operands are truthy or falsy.

*The `and` operator.*

| a | b | a and b |
|---|---|---|
| truthy | truthy | b |
| truthy | falsy | b |

Current topic:

Conversion to boolean

Topic depends on:

- ✓ Operations with list (Stage 3) 6★
- ✓ Else statement (Stage 1) 15★

Table of contents:

| | | |
|---|---|---|
| falsy | truthy | a |
| falsy | falsy | a |

*The `or` operator.*

| a | b | a or b |
|---|---|---|
| truthy | truthy | a |
| truthy | falsy | a |
| falsy | truthy | b |
| falsy | falsy | b |

*The `not` operator.*

| a | not a |
|---|---|
| truthy | `False` |
| falsy | `True` |

Now we can implement the `print_list()` function in one line:

```
def print_list(lst):
    print(lst or 'empty list')


print_list([2, 3, 4])  # [2, 3, 4]
print_list([])         # empty list
```

According to the second table, when `lst` is not empty, so it is truthy, the `or` operator returns the first operand (the list itself); when `lst` is empty, so it is falsy, the `or` operator returns the second operand (the string in our case).

Another thing to note is that when the first operand uniquely determines the result of the operation, which happens when the first operand in the `and` operation is falsy or when the first operand in the `or` operation is truthy, Python does not look at the second operand at all. For example, when we want to check if the given list `lst` is not empty and its first element is positive, we may write the following:

```
if lst and lst[0] > 0:
    ...
```

If `lst` is empty, the `lst[0] > 0` expression is invalid but it does not cause an exception because it never gets evaluated.

## §3. bool() function

Although we can use any objects in boolean expressions, there are cases when we should explicitly convert objects into real boolean values. This can be done with the `bool()` function. The `bool()` function returns `True` if the passed argument is truthy, and `False` if it is falsy.

```
print(bool(True), bool(False))    # True False
print(bool(None))                 # False
print(bool([]), bool([2, 3, 9]))  # False True
```

This function is not very common, but there are special cases when it can be useful.

## §4. When to use the bool() function?

Sometimes you need to store the result of a logical expression or even write it to a file. In this case, you would want to get `True` or `False`, not a truthy or falsy object.

Let's look at the example. You have a list of lists with integer values. You want to check if this list is not empty and its first element is not zero for each inner list. The solution is the following code:

```
def check_list(lst):
    return lst and lst[0]


lists = [[5, 9], [0, 0], []]
result = []
for lst in lists:
    result.append(check_list(lst))


print(result)  # [5, 0, []]
```

Although `5` is a truthy value and `0` and `[]` are falsy values, most likely you would prefer to get a list consisting of real boolean values: `result = [True, False, False]`. So you should explicitly convert the result of the function into a boolean value.

```
def check_list(lst):
    return bool(lst and lst[0])


lists = [[5, 9], [0, 0], []]
result = []
for lst in lists:
    result.append(check_list(lst))


print(result)  # [True, False, False]
```

When you do not have to store the result of a logical expression, there is no need to enclose the expression in the `bool()` function. For example, `if lst` is more readable than `if bool(lst)`.

## §5. Conclusions

- All objects in Python can be interpreted as boolean values.
- Some boolean operators, such as `and` and `or`, return one of the operands as a result; `not`, on the contrary, always returns a boolean value.
- It is fine to use the test for truthiness when checking if the container is empty but do it carefully.
- When you want to store the result of a logical operation, not just check if it is true or false, you should explicitly convert it to a boolean value using the `bool()` function.

Report a typo

**160** users liked this theory. **4** didn't like it. **What about you?**

Start practicing

Comments (7) Hints (1) Useful links (1) Show discussion