Bo

Bools in

3 exercises

You've mastered Bools in Python. Explore more concepts →

About Bools

Python represents true and false values with the `bool` type, which is a subtype of `int`. There are only two Boolean values in this type: `True` and `False`. These values can be assigned to a variable and combined with the **Boolean operators** (`and`, `or`, `not`):

```
>>> true_variable = True and True
>>> false_variable = True and False

>>> true_variable = False or True
>>> false_variable = False or False
```

```
>>> true_variable = not False
>>> false_variable = not True
```

Boolean operators use *short-circuit evaluation*, which means that expression on the right-hand side of the operator is only evaluated if needed.

Each of the operators has a different precedence, where `not` is evaluated before `and` and `or`. Brackets can be used to evaluate one part of the expression before the others:

```
>>>not True and True
False

>>>not (True and False)
True
```

All `boolean operators` are considered lower precedence than Python's **`comparison operators`**, such as `==`, `>`, `<`, `is` and `is not`.

Type Coercion and Truthiness

The `bool` function (**`bool()`**) converts any object to a Boolean value. By default all objects return `True` unless defined to return `False`.

A few `built-ins` are always considered `False` by definition:

- the constants `None` and `False`
- zero of any *numeric type* (`int`, `float`, `complex`, `decimal`, or `fraction`)
- empty *sequences* and *collections* (`str`, `list`, `set`, `tuple`, `dict`, `range(0)`)

```
>>>bool(None)
False

>>>bool(1)
True

>>>bool(0)
False

>>>bool([1,2,3])
True

>>>bool([])
False

>>>bool({"Pig" : 1, "Cow": 3})
True
```

```
>>>bool({})
False
```

When an object is used in a *boolean context*, it is evaluated transparently as *truthy* or *falsey* using `bool()`:

```
>>> a = "is this true?"
>>> b = []

# This will print "True", as a non-empty string is considered a "truthy" value
>>> if a:
...  print("True")

# This will print "False", as an empty list is considered a "falsey" value
>>> if not b:
...   print("False")
```

Classes may define how they are evaluated in truthy situations if they override and implement a `__bool__()` method, and/or a `__len__()` method.

How Booleans work under the hood

The `bool` type is implemented as a *sub-type* of *int*. That means that `True` is *numerically equal* to `1` and `False` is *numerically equal* to `0`. This is observable when comparing them using an

equality operator:

```
>>>1 == True
True

>>>0 == False
True
```

However, `bools` are **still different** from `ints`, as noted when comparing them using the *identity operator*, `is`:

```
>>>1 is True
False

>>>0 is False
False
```

> Note: in python >= 3.8, using a literal (such as 1, '', [], or {}) on the *left side* of `is` will raise a warning.

It is considered a **Python anti-pattern** to use the equality operator to compare a boolean variable to `True` or `False`. Instead, the identity operator `is` should be used:

```
>>> flag = True

# Not "Pythonic"
>>> if flag == True:
...     print("This works, but it's not considered Pythonic.")

# A better way
>>> if flag:
...     print("Pythonistas prefer this pattern as more Pythonic.")
```

Learn More

boolean values | boolean-operators | Truth Value Testing | bool() function | Problem Solving with Python - Boolean Data Type | Comparisons in Python | Python Anti-Patterns Comparing things to True in the Wrong Way | PEP285 - Adding a bool type

1 author
2 contributors

Edit via GitHub