



 [learn-co-curriculum](#) / [dsc-python-operators-functions-and-methods](#) Public [View license](#) 1 star  96 forks Star Watch ▾[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#) master ▾

...



hoffm386 fix objectives spacing ...

on Aug 1 ⌚ 13

[View code](#) README.md

Built-in Python Operators, Functions and Methods

Introduction

As a Data Scientist, you will spend a lot of time writing code in Python. In this lesson, we're going to introduce some features built right into the language that will allow you to perform common tasks more quickly and easily.

Objectives

You will be able to:

- Use built-in Python functions and methods
- Describe the difference between a function and a method
- Use comparison operators to compare objects
- Use logical operators to incorporate multiple conditions
- Use identity operators to confirm the identity of an object

Introducing "Objects"

Later in the course we're going to spend a good amount of time introducing and giving you hands on practice with "Object Oriented Programming" (OOP). But for now, to understand how some of the features in Python work, we're going to have to provide you with a brief introduction to some basic concepts.

A function is a piece of reusable code. If you often want to capitalize a word, you might write (or use, if someone else has written) a "function" called `capitalize()` that will take a string and make it all upper case.

An object is a collection of data and functions. It turns out that for many types of programming, putting your functions and data together into "objects" is a really useful way to organize all of your code to make it easy to keep track of.

If a function is associated to a specific object we call it a method.

Python has a number of built-in functions and various objects in Python have built-in methods.

Really understanding objects, functions, and methods will probably take a while, and it's something we'll come back to a number of times, but for now, here are the three things you need to know:

- Python comes with a range of built-in pieces of code to perform common tasks
- Some of them you use by writing their name first - e.g. `type("hello")` - those are called functions
- Others require you to take an object and "call a method on the object" e.g.
`my_name.capitalize()` - they are methods

Don't worry if it's a bit confusing for now, we'll come back to this repeatedly until it's second nature!

Python Methods

Most higher-level languages like Python have types like strings or lists that actually come built-in with some really great functionality, which we call methods. Methods are, essentially, functions that are attached or **bound to an object**. Now that sentence might not make complete sense, so, let's unpack it.

We have seen some methods and functions already. For example, we have seen and used the `type()` function and the `.title()` method. The key difference is that the `.title()` method has always been bound to a string. The `type()` function, however, can be used on a string or any other data type.

```
"hello, i am a string.".title() # "Hello, I Am A String."
```

```
type("hello, i am a string.") # str
```

To illustrate even more clearly, try running the next cell. We will find that Python doesn't know what `title()` is when it is not attached to a string. This is because it is a **method** that is **bound** to the string.

```
title("hello")
```

Don't worry too much about the differences between the two. For now, just know that methods will always need an object or piece of data to be called on, like a string, list, or dictionary. Now let's take a look at a few of these frequently used methods.

First, let's look at the `.upper()`, `.lower()`, `.capitalize()` methods, which are all **string methods**, meaning they are only called on strings.

- `.upper()` is used to make all characters in a string uppercased
- `.lower()` is used to make all characters in a string lowercased
- `.capitalize()` is used to make only the **first** character in a string uppercased

```
print("hello, im uppercased".upper())  
print("HELLO, IM LOWERCASED".lower())  
print("hello, im capitalized".capitalize())
```

Next, let's look at some **list methods**:

- `.append(ELEMENT)` is used to add a given element to the end of a list.
- `.pop()` is used to remove the last element from the list (or if an index is given, it removes the element at that index).
- `.extend([SECOND_LIST])` is used to add all elements from a second list to the first list.

It is important to note the **return value** from each of these methods is not the resulting list. These operations **alter** the original list on which we are operating.

```
list_append = [1,2,3,4]
list_append.append(5)
print(list_append)
```

```
list_pop = [4,5,6,7]
list_pop.pop()
print(list_pop)
```

```
list_one = [1,2,3]
list_two = [4,5,6]
list_one.extend(list_two)
print(list_one)
```

Finally, we'll look at a few **dictionary methods**:

- `.keys()` is used to return a list-like `dict_keys` object with the name of each key in the dictionary
- `.values()` is used to return a list-like `dict_values` object with the values in the dictionary

```
dictionary = {'name-key': 'example-dict', 'key_2': 'value_2', 'num_keys': 3}
print(dictionary.keys())
print(dictionary.values())
```

Note: If we wanted to see all built-in methods for a data type, we can call Python's `dir()` function as shown below. We just need to give the `dir()` function the data type we want to look at (i.e. `str`, `dict`, `list`, `int` etc.)

```
dir(str)
```

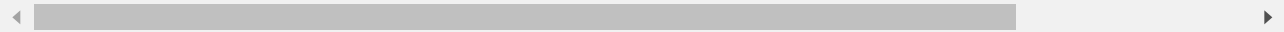
Python Functions

Next, we will talk about some of Python's built-in **functions**. Unlike methods, functions are not bound to any particular object or type of object. They can be called by themselves, however, they typically require an argument. Let's take a look at an example before we get too far into how functions work.

The `print()` function takes in as many arguments as we want. It then returns each argument *stringified* or the string form of each argument. It can also be executed without any argument, but we will just not have any return value.

```
print()
```

```
print(["this", "is", "a", "list"], {"this-is": 'a-dictionary'}, "this is a string an
```



The `type()` function takes in one argument, any piece of data, and it returns the **type** of that data. So, if the argument is a dictionary (`{}`), we will get the return value of `dict` .

```
type({})
```

```
type("")
```

The `len()` function takes in a collection, which is generally an object we can iterate through, like a dictionary or a list, and it returns the number of elements contained within that collection. In the case of a dictionary, it returns the number of key, value pairs.

```
len({"this-is": 'a-dictionary', "with-two": "key value pairs"})
```

```
len(["this", "is", 1, "list", "with", 7, "elements"])
```

The `sum()` function operates on lists that contain **only numbers** and returns the sum of all the numbers within the list.

```
sum([10, 11, 13, 14, 9, 14.5])
```

The `max()` and `min()` functions operate on collections that **contain elements of certain compatible data types**, which can be compared using a comparison operator like `<` , which we will explore later. Generally, we will be using these functions on lists with either all **numbers** or all **strings**.

```
min(["this", "is", "a", "list"])
```

```
max([10, 11, 13, 14, 9, 14.5])
```

Python Operators

What Are Comparison Operators?

Comparison operators (or Relational operators) take two elements and compare their values and then return a value that is either True or False. In Python, comparison operators are:

```
== # tests equality between two elements
!= # tests inequality between two elements
<, >, <=, >= # each tests the value between two elements
```

Perhaps the last line's operators are a little more familiar because we've seen these operators in math classes. But the first two might be a bit more confusing, so, let's dive into those first.

The double equals operator (`==`) is testing whether the value of the first element is equal to that of the second element (e.g. `element_1 == element_2`).

```
False == True # returns False
False == False # returns True
10 == 20 # returns False
10 == 10 # returns True
"hi" == "HI" # returns False
"heLlO" == "heLlO" # returns True
```

The bang (exclamation point) equals operator (`!=`) is testing whether the value of the first element is **NOT** equal to that of the second element (e.g. `element_1 != element_2`).

```
True != True # returns False
True != False # returns True
10 != 20 # returns True
10 != 10 # returns False
"hi" != "HI" # returns True
"heLlO" != "heLlO" # returns False
```

Now onto the third grouping of comparison operators. The greater than (>), less than (<), greater than or equal to (>=) and less than or equal (<=) to operators also only return `True` or `False` as the output.

```
True > True # False
True >= True # True
10 <= 10 # True
7 < 7 # False
10 < 100 # True
100 > 101 # False
```

We can even compare strings to see which is alphabetically greater or less. A string is greater if it comes after another string alphabetically (or if its ASCII value is greater). An important note is that capital letters have lower ASCII values, which means that when you make comparisons, uppercase letters will always be less than lowercase letters. For example, the ASCII alphabet would look something like: A, B, C ... X, Y, Z ... a, b, c ... x, y, z with A having the lowest ASCII value and z having the highest ASCII value.

```
"APPLE" < "apple" # True
"aaron" > "alexa" # False
"Terrance" > "Teresa" # True
"SAME" == "SAME" # True
```

So, when comparing strings to other strings, it is important to be cognizant of whether the comparison should be case sensitive or not.

How Are Comparison Operators Used?

We've already seen some examples of this above. Essentially, we use comparison operators to *compare* two things. So, the most basic structure is two elements with an operator in between. The return value will be `True` or `False`. The best way to get comfortable with comparison operators is through practice, so, let's take a look at some examples:

```
print('1.', True != True) # False
print('2.', False == True) # False
print('3.', 10 == "10") # False
print('4.', "hi" != "HI") # True
print('5.', type(0) == int) # True
print('6.', ["hi"] == ["hi"]) # True
print('7.', "Thomas" != "Samantha") # True
```

Above we can see examples using both the `!=` and `==` operators to compare elements of several data types and values. Remember that these operators are testing for the value of each element.

Next, we will look at examples using the greater than (`>`), less than (`<`), greater than or equal to (`>=`) and less than or equal (`<=`) to operators.

```
print('1.', True > True) # False
print('2.', True >= True) # True
print('3.', 10 <= 10) # True
print('4.', 7 < 7) # False
print('5.', 10 < 100) # True
print('6.', 100 > 101) # False
print('7.', "APPLE" < "apple") # True
print('8.', "aaron" > "alexa") # False
print('9.', "Terrance" > "Teresa") # True
```

Other Types of Operators - Logical Operators

The next group of operators are logical operators. They provide a means for creating more complex logical phrases by combining multiple truthy/falsy values. The operators are:

`and`

Evaluates 2 elements x and y; x is evaluated first. If x is falsy, x is returned.

`or`

Evaluates 2 elements x and y; x is evaluated first. If x is truthy, its value is r

`not`

returns a boolean value that is the opposite of the truthy/falsy value of the eler

Let's see these in action:

*try to reason what the values will be before running the code - **write** your answers in a comment next to the print statement*

```
print("1.", 2 and 0) #
print("2.", False and 2) #
print("3.", True and 2) #
print("4.", 2 and 3) #
```



```
print("5.", 2 or []) #
print("6.", 0 or []) #
print("7.", not False) #
print("8.", not True) #
print("9.", not []) #
print("10.", not 0) #
print("11.", not 100) #
```

Other Types of Operators - Identity Operators

The next type of operators are identity operators, `is` and `is not`. They check to see if one element **is** or **is not** the other element. This is similar to `!=` and `==`. However there is one key difference. The `!=` and `==` check to see if the value of each element is the same, however, the `is` and `is not` operators check to see if the elements are the same element.

Let's check it out:

again, try to find the correct return value for each example and write the answer in a comment next to the print statement

```
x = {'name': "example"}
b = x
c = {'name': "example"}
print("1.", {} is {}) #
print("1A.", {} == {}) #
print("2.", [] is []) #
print("3.", "Hi" is "Hi") #
print("4.", ["same"] is ["same"]) #
print("4A.", ["same"] == ["same"]) #
print("5.", 9 is not 10) #
print("6.", x is b) #
print("7.", b is c) #
print("7A.", b == c) #
print("8.", x is not c) #
```

As we can see, the `is` and `is not` operators are checking to see if the objects are exactly the same object in memory. However, the `==` and `!=` operators are simply checking to see if the value of each element is the same. This will become clearer as we learn more about how Python stores data.

All objects are stored in a specific place in memory, we can think of this as an address on the computer, so, all objects will have their own unique address. A **variable**, like `x`, is a **reference** to that object and not the object itself. So, when we use the `is` or `is not` operator, we are checking to see if the address of the object is the same as another object. When we use the `==` or `!=` operators, we are checking to see if they are basically equal in value, irrespective of whether they are the same or two different objects.

BONUS: Ternary Operator

The next operator is the Ternary. It is a bit more of a complicated operator, but it can be very useful when you would like to decide which value to assign to a variable. Ternaries are good for one-line conditions, but anything more complex makes ternary operators quite difficult to read. Let's check it out:

```
my_condition = True
value = 10 if my_condition else 1000
print(value)
```

```
# let's say we are receiving two variables with different
# values and we want to assign the higher value to a new variable
x = 12
y = 20
new_variable = x if x > y else y
# here we are saying, take the value of x if it is greater than the value of y. else
# since x > y evaluates to false, the ternary returns the value of the variable after
print(new_variable)
```

Summary

In this lesson, we covered a lot of material, so don't worry if it seems overwhelming right now. First we looked at objects and then built-in methods and functions in Python. They help us greatly reduce the amount of code we write while also increasing the readability and efficiency of our code.

Next we looked at operators in Python. Operators are fundamental tools in many languages that provide a succinct way to compare multiple elements. Comparison operators return boolean values and compare the value between two elements. Logical operators compare the truthiness and falsiness of two elements and either return one of the elements or a boolean value. Identity operators compare two elements for their equality, that is whether they are the same object or not, and return a boolean value. Lastly, ternary operators are used to assign a value to a variable. They use an `if` statement and another operator to compare two values and return one of two values, which is used to assign the value of a variable.

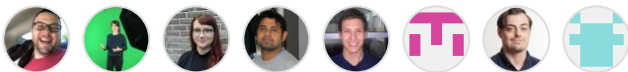
Releases

No releases published

Packages

No packages published

Contributors 8



Languages

● Jupyter Notebook 100.0%