Functions are objects

1. Functions as objects

00:00 - 00:13

In this chapter, you are going to learn about decorators, a powerful way of modifying the behavior of functions. But first, we need to build up some foundational concepts that will make decorators easier to understand.

2. Functions are just another type of object

00:13 - 00:31

The main thing you should take away from this lesson is that functions are just like any other object in Python. They are not fundamentally different from lists, dictionaries, DataFrames, strings, integers, floats, modules, or anything else in Python.

3. Functions as variables

00:31 - 01:01

And because functions are just another type of object, you can do anything to or with them that you would do with any other kind of object. You can take a function and assign it to a variable, like "x". Then, if you wanted to, you could call x() instead of my_function(). It doesn't have to be a function you defined, either. If you felt so inclined, you could assign the print() function to PrintyMcPrintface, and use it as your print() function.

4. Lists and dictionaries of functions

01:01 - 01:38

You can also add functions to a list or dictionary. Here, we've added the functions my_function(), open(), and print() to the list "list_of_functions". We can call an element of the list, and pass it arguments. Since the third element of the list is the print() function, it prints the string argument to the console. Below that, we've added the same three functions to a dictionary, under the keys "func1", "func2", and "func3". Since the print() function is stored under the key "func3", we can reference it and use it as if we were calling the function directly.

5. Referencing a function

01:38 - 02:06

Notice that when you assign a function to a variable, you do not include the parentheses after the function name. This is a subtle but very important distinction. When you type my_function() with the parentheses, you are calling that function. It evaluates to the value that the function returns. However, when you type "my_function" without the parentheses, you are referencing the function itself. It evaluates to a function object.

6. Functions as arguments

02:06 - 02:42

Here's where things get really fun. Since a function is just an object like anything else in Python, you can pass one as an argument to another function. The has_docstring() function checks to see whether the function that is passed to it has a docstring or not. We could define these two functions, no() and yes(), and pass them as arguments to the has_docstring() function. Since the no() function doesn't have a docstring, the has_docstring() function returns False. Likewise, has_docstring() returns True for the yes() function.

7. Defining a function inside another function

02:42 - 02:55

Functions can also be defined inside other functions. These kinds of functions are called nested functions, although you may also hear them called inner functions, helper functions, or child functions.

8. Defining a function inside another function

02:55 - 03:12

A nested function can make your code easier to read. In this example, if x and y are within some bounds, foo() prints x times y. We can make that if statement easier to read by defining an in_range() function.

9. Functions as return values

03:12 - 03:41

There's also nothing stopping us from returning a function. For instance, the function get_function() creates a new function, print_me(), and then returns it. If we assign the result of calling get_function() to the variable "new_func", we are assigning the return value, "print_me()" to "new_func". We can then call new_func() as if it were the print_me() function.

10. Let's practice!

03:41 - 03:58

The way that Python treats everything as an object is one of my favorite things about the language, and it gives you the ability to do a lot of really complex things. In the coming lessons, we'll dig into what this ability allows us to do. For now, you can check your understanding with a few exercises.