**Closures**

**1. Closures**

00:00 - 00:06

The last topic I need to explain before discussing decorators is how closures work in Python.

**2. Attaching nonlocal variables to nested functions**

00:06 - 01:22

A closure in Python is a tuple of variables that are no longer in scope, but that a function needs in order to run. Let's explain this with an example. The function foo() defines a nested function bar() that prints the value of "a". foo() returns this new function, so when we say "func = foo()" we are assigning the bar() function to the variable "func". Now what happens when we call func()? As expected, it prints the value of variable "a", which is 5. But wait a minute, how does function "func()" know anything about variable "a"? "a" is defined in foo()'s scope, not bar()'s. You would think that "a" would not be observable outside of the scope of foo(). That's where closures come in. When foo() returned the new bar() function, Python helpfully attached any nonlocal variable that bar() was going to need to the function object. Those variables get stored in a tuple in the "__closure__" attribute of the function. The closure for "func" has one variable, and you can view the value of that variable by accessing the "cell_contents" of the item.

**3. Closures and deletion**

01:22 - 02:08

Let's examine this bit of code. Here, x is defined in the global scope. foo() creates a function bar() that prints whatever argument was passed to foo(). When we call foo() and assign the result to "my_func", we pass in "x". So, as expected, calling my_func() prints the value of x. Now let's delete x and call my_func() again. What do you think will happen this time? If you guessed that we would still print 25, then you are correct. That's because foo()'s "value" argument gets added to the closure attached to the new "my_func" function. So even though x doesn't exist anymore, the value persists in its closure.

**4. Closures and overwriting**

02:08 - 02:35

Notice that nothing changes if we overwrite "x" instead of deleting it. Here we've passed x into foo() and then assigned the new function to the variable x. The old value of "x", 25, is still stored in the new function's closure, even though the new function is now stored in the "x" variable. This is going to be important to remember when we talk about decorators in the next lesson.

**5. Definitions - nested function**

02:35 - 02:49

Let's go over some of the key concepts again to be sure you understand. A nested function is a function defined inside another function. We'll sometimes refer to the outer function as the parent and the nested function as the child.

**6. Definitions - nonlocal variables**

02:49 - 02:58

A nonlocal variable is any variable that gets defined in the parent function's scope, and that gets used by the child function.

**7. Definitions - closure**

02:58 - 03:11

And finally, a closure is Python's way of attaching nonlocal variables to a returned function so that the function can operate even when it is called outside of its parent's scope.

**8. Why does all of this matter?**

03:11 - 03:37

We've gone pretty deep into the internals of how Python works, and you must be wondering, "Why does all of this matter?" Well, in the next lesson we'll finally get to talk about decorators. In order to work, decorators have to make use of all of these concepts: functions as objects, nested functions, nonlocal scope, and closures. Now that you have a firm foundation to build on, understanding how decorators work should be easy.

**9. Let's practice!**

03:37 - 03:47

Before we move on though, try out a few practice problems to make sure you understand how closures work.