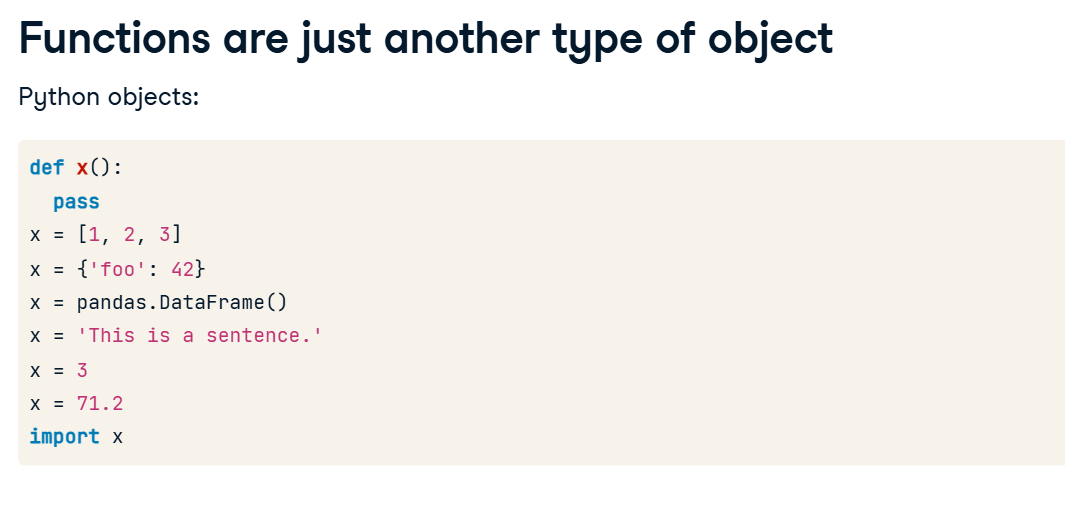
**Functions as objects**

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.

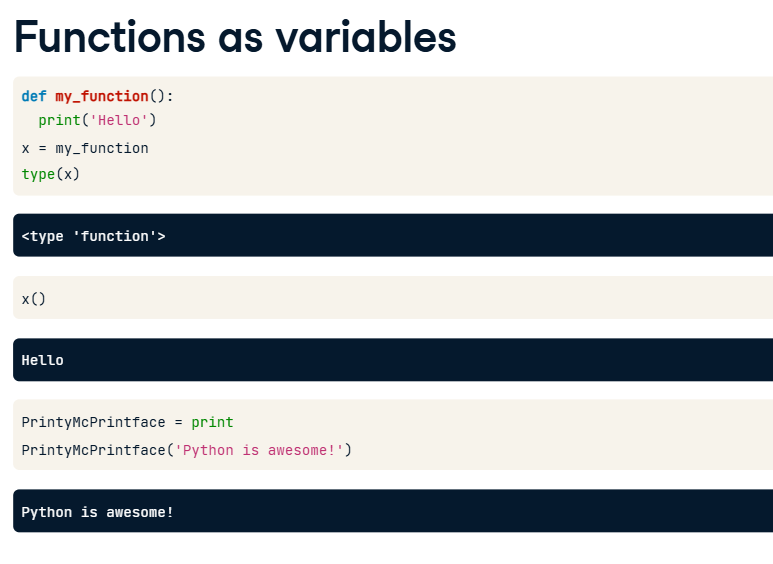
**Functions are just another type of object**

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.



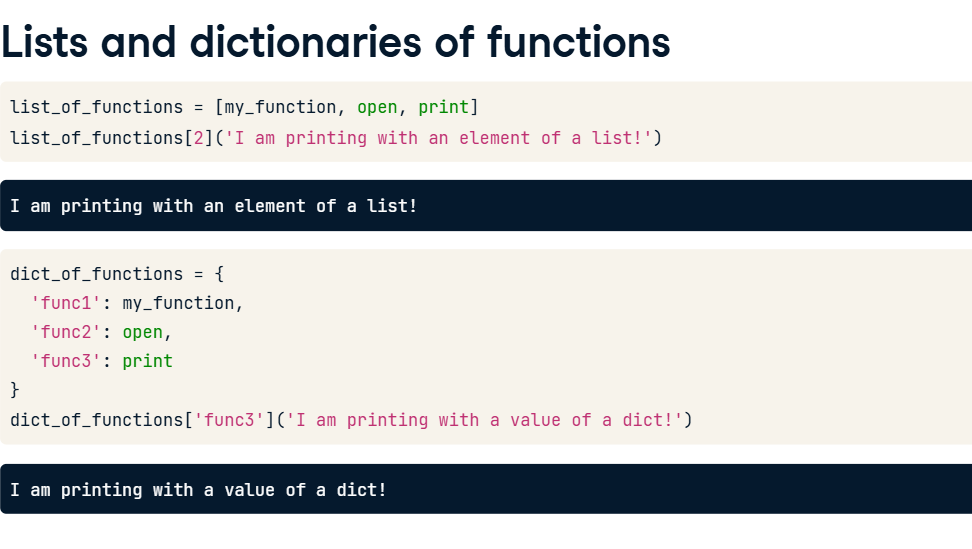
**Functions as variables**

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.



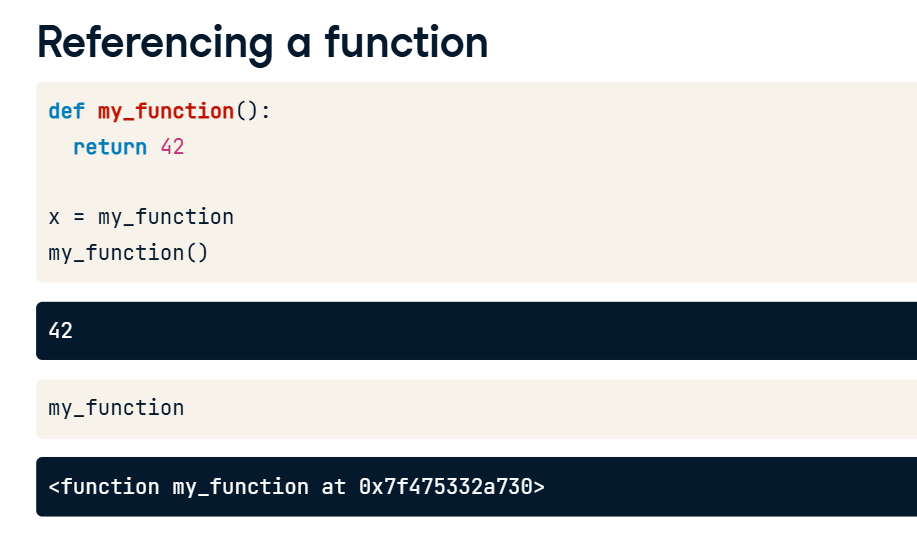
**Lists and dictionaries of functions**

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.



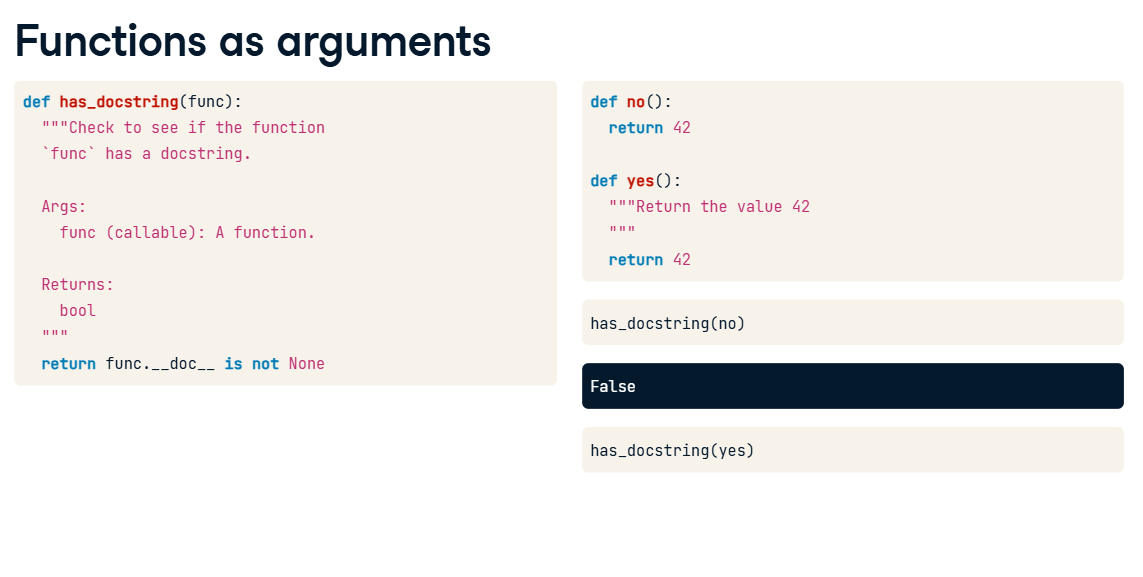
**Referencing a function**

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.



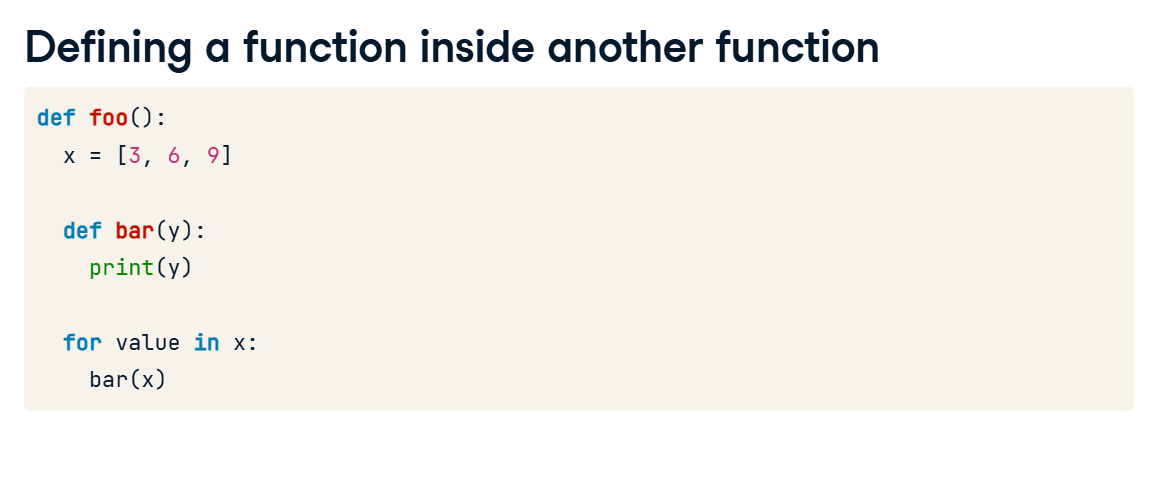
**Functions as arguments**

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.

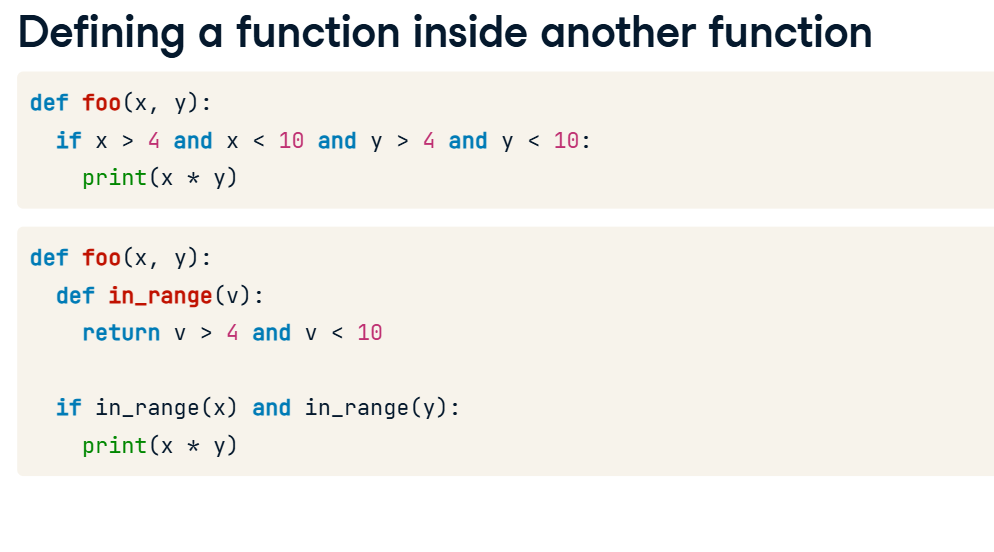


**7. Defining a function inside another function**

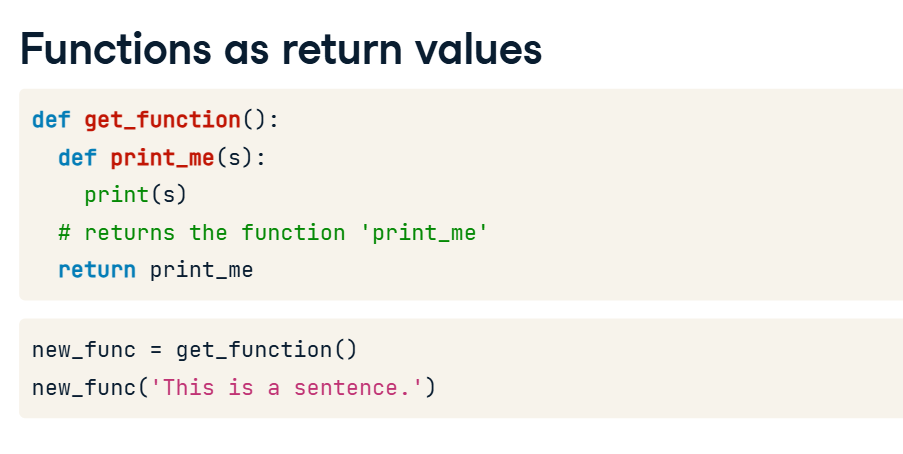
Functions can also be defined inside other functions. These kinds of functions are called nested functions, although you may also hear them called inner functions, helper functions, or child functions.



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.



**Functions as return values**

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. 

**Scope**

Before we can dig into decorators, we must understand how scope works in Python. Scope determines which variables can be accessed at different points in your code.

**Names**

Names are very useful things, both in Python and in the real world. For instance, this is Tom.

And this is Janelle. When we say "Tom", we know we are talking about the person on the left, and when we say "Janelle", we know we are talking about the person on the right.

**Scope**

If Janelle says, "Tom didn't go to work yesterday," we can be fairly sure she is talking about the Tom standing next to her,

And not some Tom in a different country. This is sort of how scope works in programming languages like Python.

Python has names too—variable names. When we say "print(x)" here, Python knows we mean the x that we just defined. What happens if we redefine x inside the function foo() though? In foo()'s print() statement, do we mean the x that equals 42 or the x that equals 7? Python applies the same logic we applied with Tom and Janelle and assumes we mean the x that was defined right there in the function. However, there is no y defined in the function foo(), so it looks outside the function for a definition when asked to print y. Note that setting x equal to 42 inside the function foo() doesn't change the value of x that we set earlier outside of the function.

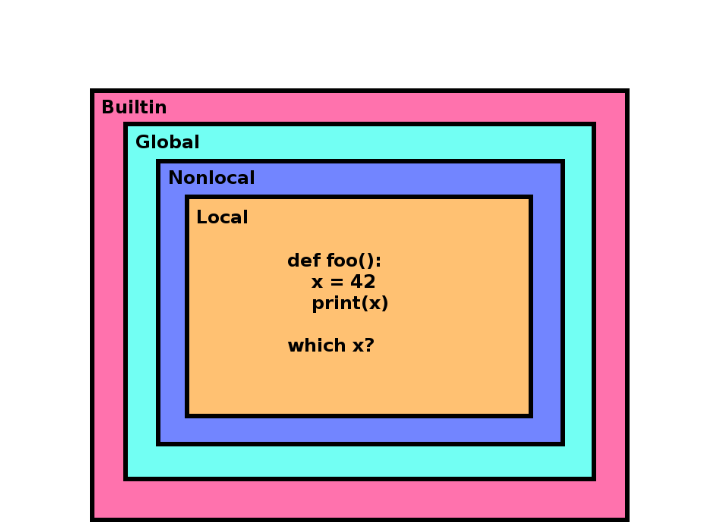
Python has to have strict rules about which variable you are referring to when using a particular variable name. So when we typed print(x) in the function foo(), the interpreter had to follow those rules to determine which x we meant.

First, the interpreter looks in the local scope. When you are inside a function, the local scope is made up of the arguments and any variables defined inside the function.

If the interpreter can't find the variable in the local scope, it expands its search to the global scope. These are the things defined outside the function.

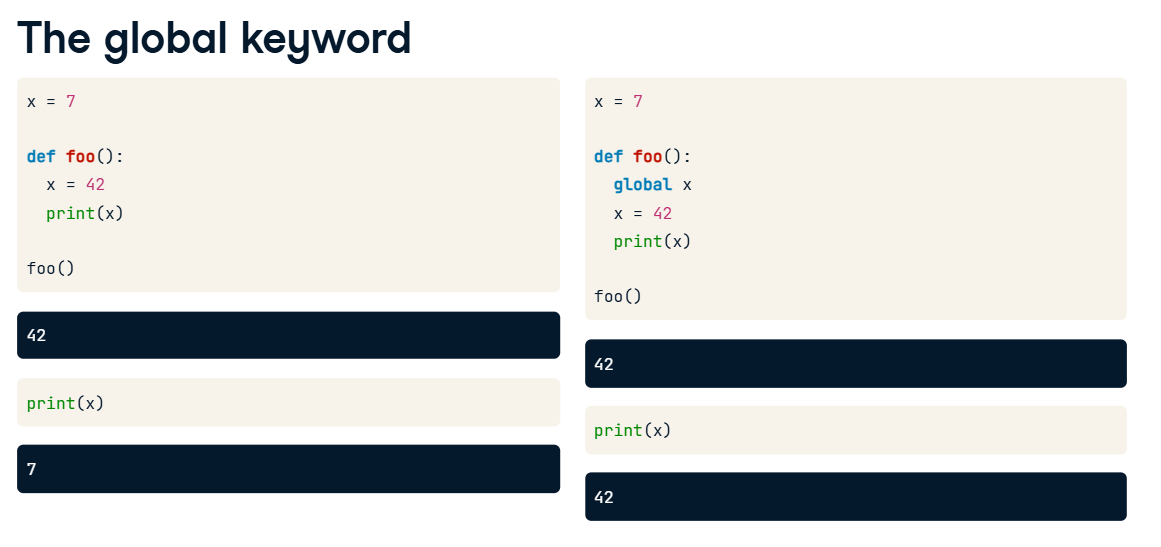
Finally, if it can't find the thing it is looking for in the global scope, the interpreter checks the builtin scope. These are things that are always available in Python. For instance, the print() function is in the builtin scope, which is why we are able to use it in our foo() function.

I actually skipped a level in that diagram. In the case of nested functions, where one function is defined inside another function, Python will check the scope of the parent function before checking the global scope. This is called the nonlocal scope to show that it is not the local scope of the child function and not the global scope.



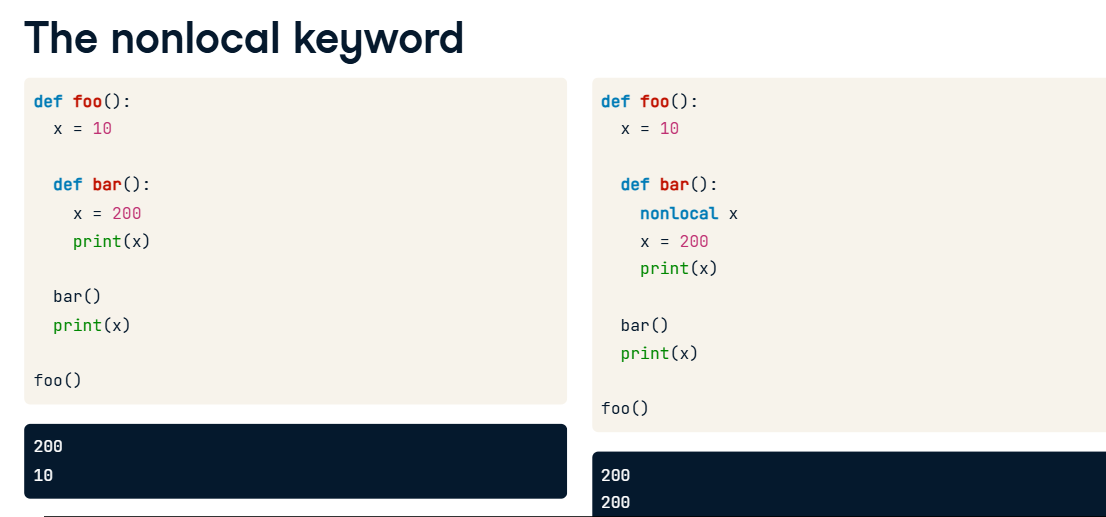
**The global keyword**

Note that Python only gives you read access to variables defined outside of your current scope. In foo() when we set x equal to 42, Python assumed we wanted a new variable in the local scope, not the x in the global scope. If what we had really wanted was to change the value of x in the global scope, then we have to declare that we mean the global x by using the global keyword. Notice that when we print x after calling foo() now, it prints 42 instead of 7 like it used to. However, you should try to avoid using global variables like this if possible, because it can make testing and debugging harder.



**The nonlocal keyword**

And if we ever want to modify a variable that is defined in the nonlocal scope, we have to use the "nonlocal" keyword. It works exactly the same as the "global" keyword, but it is used when you are inside a nested function, and you want to update a variable that is defined inside your parent function.



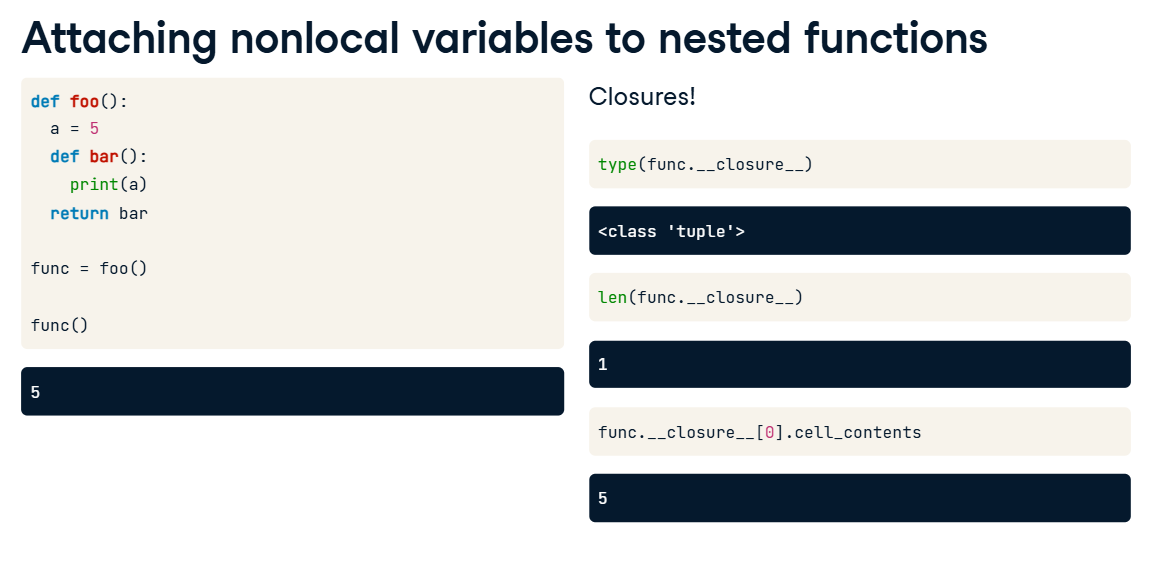
**Closures**

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

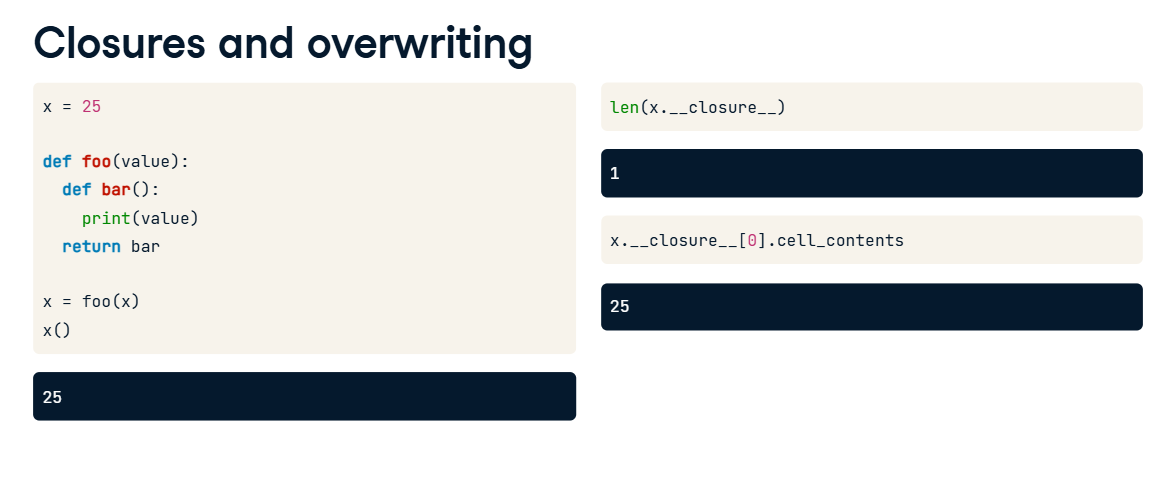
Let's examine this bit of code. Here, x is defined in the global scope. foo() creates a function bar() that prints whatever argument was passed to foo(). When we call foo() and assign the result to "my\_func", we pass in "x". So, as expected, calling my\_func() prints the value of x. Now let's delete 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**

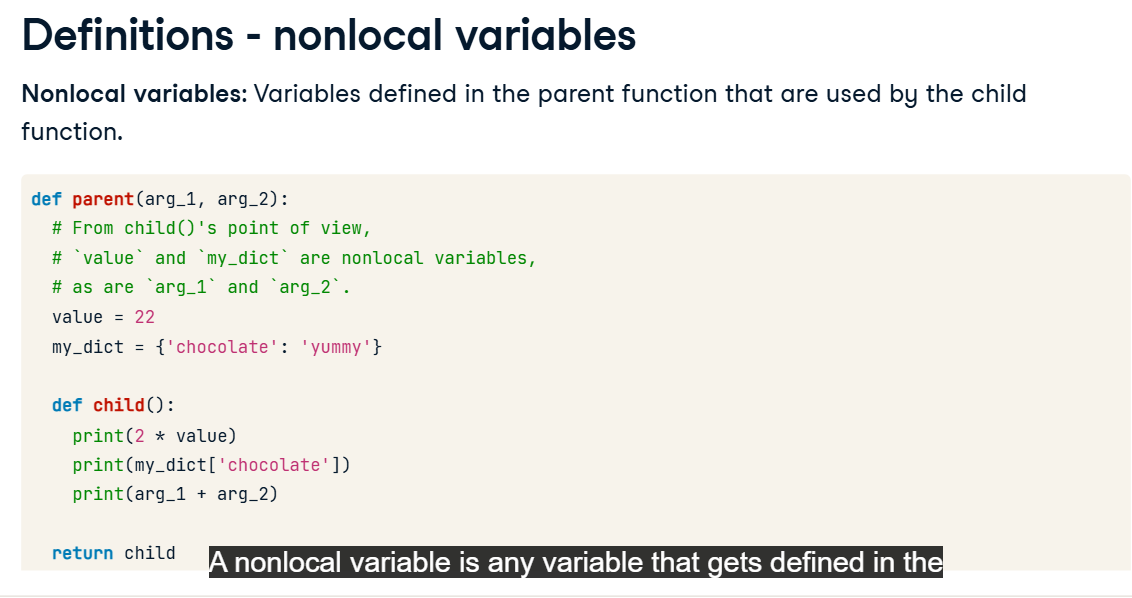
02:08 - 02:35

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.



**Definitions - nested function**

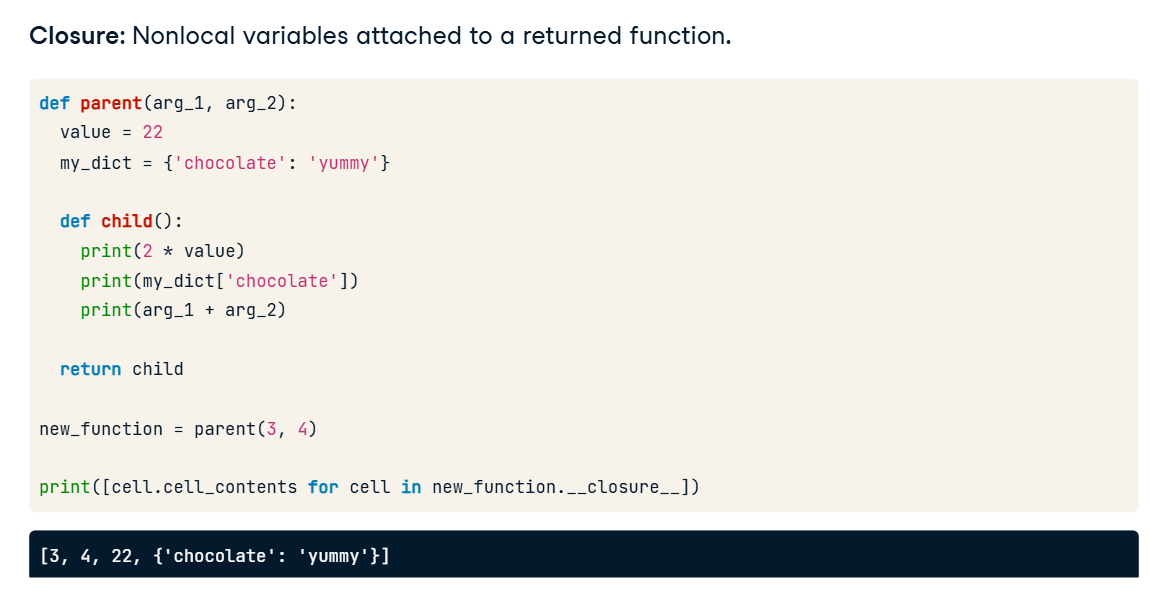
Let's go over some of the key concepts again to be sure you understand. A nested function is a function defined inside another function. We'll sometimes refer to the outer function as the parent and the nested function as the child.



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

**Definitions - closure**

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.



**Why does all of this matter?**

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.

**Decorators**

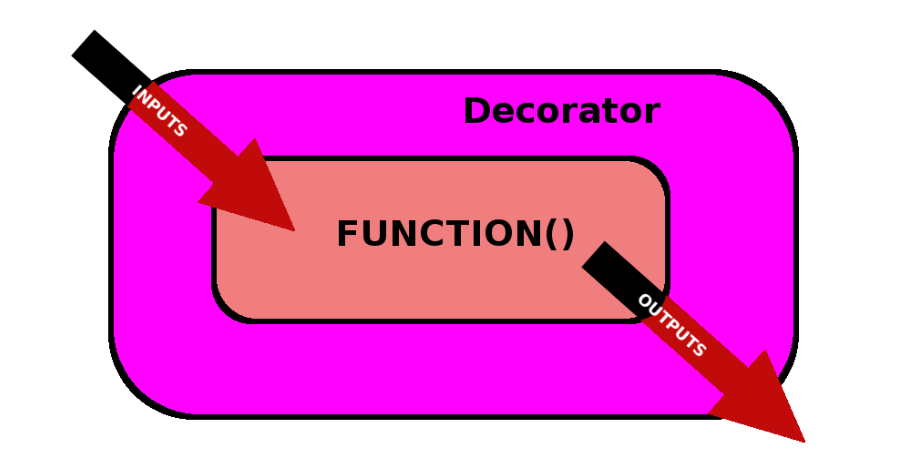
Now that you know functions can be passed around as variables, and you understand scope and closures, we can talk about decorators.

**Functions**

So what is a decorator? Let's say you have a function that takes some inputs and returns some outputs.

**Decorators**

A decorator is a wrapper that you can place around a function that changes that function's behavior.

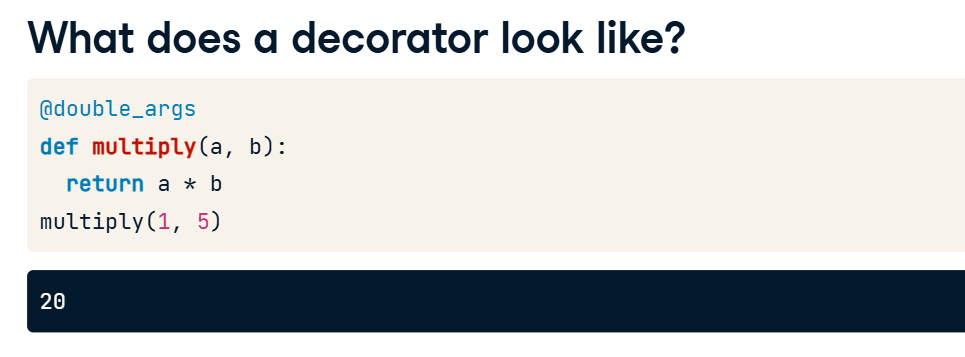


**Modify inputs**

You can modify the inputs, modify the outputs, or even change the behavior of the function itself.

**What does a decorator look like?**

You may have seen decorators in Python before. When you use them, you type the "@" symbol followed by the decorator's name on the line directly above the function you are decorating. Here, the "double\_args" decorator modifies the behavior of the multiply() function. double\_args is a decorator that multiplies every argument by two before passing them to the decorated function. So 1 times 5 becomes 2 times 10, which equals 20. That seems kind of magical that we can alter the behavior of functions, so let's peel back the layers and see how it works. We will build the double\_args decorator together in this lesson.



**8. The double\_args decorator**

01:12 - 01:50

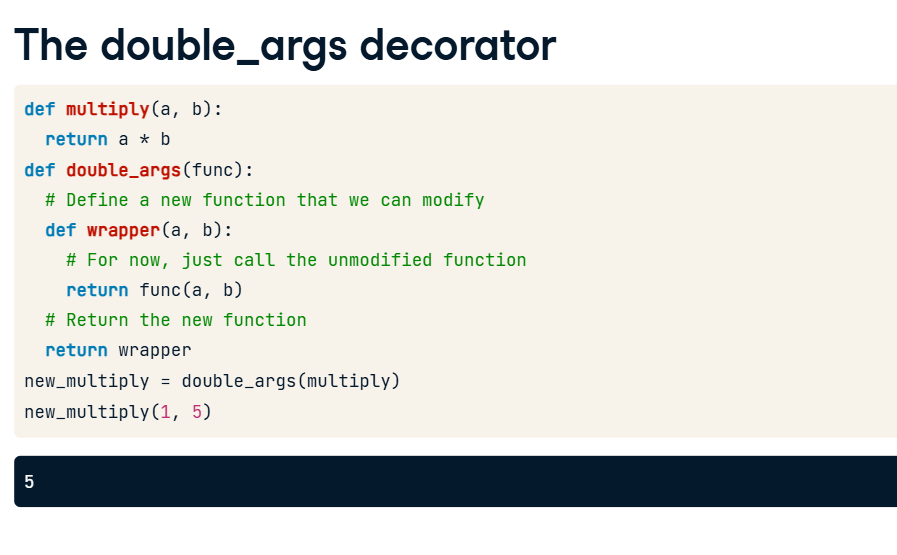
Let's continue to use the multiply() function as the function we are decorating. Now, decorators are just functions that take a function as an argument and return a modified version of that function. To start off, let's not have double\_args modify anything. It just takes a function and immediately returns it. If we call this version of double\_args() that does nothing and pass it the multiply function and then assign the result to the variable "new\_multiply", then we can call new\_multiply(1, 5) and get the same value we would have gotten from multiply(1, 5).



**9. The double\_args decorator**

01:50 - 02:49

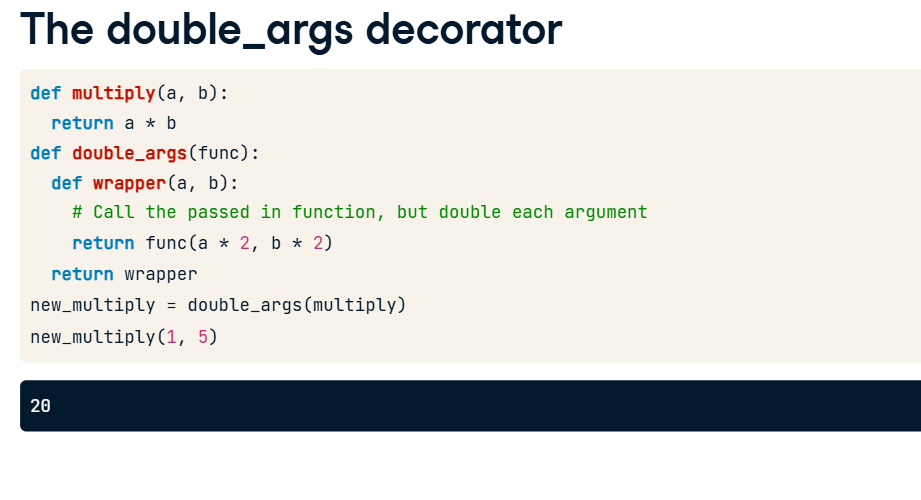
In order for your decorator to return a modified function, it is usually helpful for it to define a new function to return. We'll call that nested function "wrapper()". All wrapper() does is take two arguments and passes them on to whatever function was passed to double\_args() in the first place. If double\_args() then returns the new wrapper() function, the return value acts exactly the same as whatever function was passed to double\_args(), assuming that the function passed to double\_args() also takes exactly two arguments. So, double\_args() is still not doing anything to actually modify the function it is decorating. Once again, we'll pass multiply() to double\_args() and assign the result to new\_multiply(). If we then call new\_multiply(), which is now equal to the wrapper() function, wrapper() calls multiply() because it is the function that was passed to double\_args(). So wrapper() calls multiply() with the arguments 1 and 5, which returns 5.



**10. The double\_args decorator**

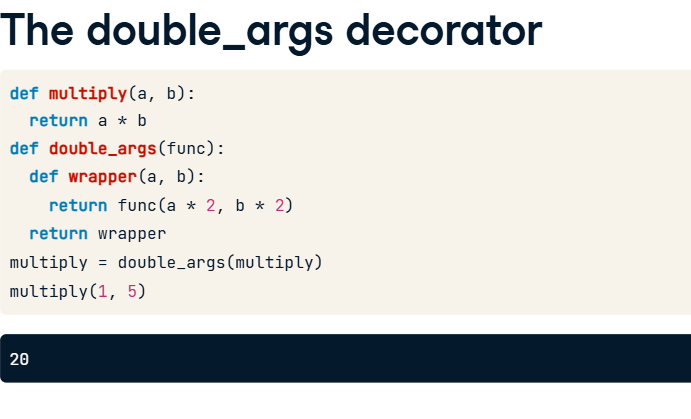
02:49 - 03:34

Now let's actually modify the function our decorator is decorating. This time, wrapper() will still call whatever function is passed to double\_args(), but it will double every argument when it calls the original function. See how it calls func() with a times 2 and b times 2? As usual, we will call double\_args() on the multiply() function and assign the result to new\_multiply(). Now, what happens when we call new\_multiply() with 1 and 5 as arguments? Well, new\_multiply() is equal to wrapper(), which calls multiply() after doubling each argument. So 1 becomes 2 and 5 becomes 10, giving us 2 times 10, which equals 20.



**The double\_args decorator**

We're almost there. This time, instead of assigning the new function to "new\_multiply", we're going to overwrite the "multiply" variable. Now calling multiply() with arguments 1 and 5 gives us 20 instead of 5. Remember that we can do this because Python stores the original multiply function in the new function's closure.



**Decorator syntax**

When I first showed you the double\_args() decorator at the beginning of this lecture, I used "@double\_args" on the line before the definition of multiply(). This is just a Python convenience for saying "multiply" equals the value returned by calling double\_args() with "multiply" as the only argument. The code shown here on the left is exactly equivalent to the code on the right.

