SCOPE

1. Scope

00:00 - 00:10

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.

2. Names

00:10 - 00:18

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

3. Names

00:18 - 00:27

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.

4. Scope

00:27 - 00:35

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,

5. Scope

00:35 - 00:42

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

6. Scope

00:42 - 01:31

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.

7. Scope

01:31 - 01:47

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.

8. Scope

01:47 - 01:57

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.

9. Scope

01:57 - 02:07

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.

10. Scope

02:07 - 02:24

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.

11. Scope

02:24 - 02:45

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.

12. The global keyword

02:45 - 03:27

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.

13. The nonlocal keyword

03:27 - 03:45

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.

14. Let's practice!

03:45 - 03:52

Try a few exercises to make sure you understand how scope works before moving on to the next lesson.