Module 5
Video Transcripts

Video 1: Functions Review (3:42)

Before you learn some more advanced topics about functions, let's do a quick review of what you should already know. Python comes with a lot of built-in functions. Some functions, you should already be familiar with are str(), int(), float(), bool(), print(), max(), min(), sum(), abs(), round(), type(), and del(). We learned that functions are followed by parentheses, and in the parentheses, you specify the arguments. The arguments are inputs that the function performs some calculations on and then produces an output.

We also learned that additional functions could be imported from packages. By now you should be familiar with Numpy and pandas. When you call a function from a package, you need to refer to the package or the package alias with a period before the functioning. Numpy includes the array function that converts its input into an array. You can also use Numpy to find the mean, median, and correlation. An important function we learned in pandas is 'pd.DataFrame', which converts its input into a DataFrame.

And functions from packages, we learned about writing our own functions called user-defined function. We start with the function header. This includes the keyword 'def', followed by the name you want to give your function and followed by parentheses. In the parentheses, specify the function parameters, if there are any. Finally, end the function header with a colon. Below the function header, you have the function body.

This is the code that tells you what the function will do, and this indents at four spaces under the function header. We learned that we can create a function with zero, one or multiple parameters. If we have more than one parameter, we need to be aware of the order in which we enter our arguments. The first argument will correspond to the first parameter; the second argument will correspond to the second parameter, and so on.

For example, in this function, the first argument we enter will correspond to greeting, and the second argument we enter will correspond to name. Here we print out '('Good morning, ', 'George!')'. If we answer the arguments in reverse, it will say '("George!", "Good morning, ")'. If we want to enter the arguments in an order different than what specified in the function definition, we can do that so long as we explicitly state which argument goes with which parameter using an equal sign like this.

Here even though we entered 'George' first, our output will still be 'Good morning, George!'. To review, we discussed some built-in functions that you should already be familiar with, along with functions from the Numpy and pandas packages that you can use to perform some quick calculations and create DataFrames. We also discussed the structure of a function which starts out with the function header and ends with the function body. And we talked about what to do if you have multiple parameters in your function header.

****

Video 2: Global Scope vs Local Scope (4:24)

You should by now be familiar with writing user-defined functions. Now, we'll build on that knowledge with a discussion of scope. We'll first define scope and the three different types of scope, and then examine why scope matters in the context of writing user-defined functions. Scope refers to the place that you can see your access and name. A name could be the name of a variable, a function, a method, or anything else within Python. There are three types of scope, global scope, local scope, and built-in scope.

A name is in the global scope if it is defined in the main body of the script. It's in the local scope if it's defined within a function. And it's in the built-in scope if it comes standard with Python. Like many functions, you should already be familiar with like print, max, min, del, and so on. This is important in the context of user-defined functions because many of your functions may change or create variables within the function.

But then those same variables aren't successful outside of the function, or you can create a variable in the global scope and change it with a function; when you look at the variable again within the global scope, you'll see that the changes weren't saved. In this lesson, we'll see how this works and discuss techniques we can use to avoid scope problems. For example, let's look at this user-defined function which adds '2' to 'j' and prints it out.

We can test this out with '5' as the argument. This prints out '7' because 5 plus 2 is 7. That '7' gets assigned to 'j'. So, when 'j' is printed out, the output is '7'. Great, so 'j' equals '7', right? Well, not exactly. If you try to 'print(j)' now, you'll get an error. That's because 'j' is defined in the local scope, meaning in the function. So, it's not accessible in the global scope, meaning the main body of the script. In the global scope, 'j' doesn't even exist. So, let's try something else. Let's define 'j' in the global scope. Let's say 'j = 10'.

Let's run the function with '5' again. You might expect to get 12 because 2 plus 10 is 12. But you'll get '7'. That's because Python first searches in the local scope and then in the global scope. In the local scope, we're entering '5'. Whereas, in the global scope, 'j' is equal to '10'. Now, if Python can't find the value of a variable within the local scope, it's gonna search in the global scope. So, if you run 'add_two(j)', now we'll get '12'.

But what if you want to use the value of a name in your global scope within a function? You can use the keyword 'global' like this. Simply type 'global' followed by the variable name at the start of your function. Now, it will refer to whatever value of 'p' is defined in the global scope. In this case, it's '4'. So, the function will output '6'. Keep in mind, based on our function definition; this actually changes the value of 'p'.

So, we run it again now, and we'll see that it does 6 plus 2 and outputs '8'. Then '10', then '12' and so on. To review, there are three types of scope, global, local, and built-in. A variable defined in a function is in a local scope, and a variable defined in the main body of the script is in the global scope. When you run a Python script, the script will first search for a name in the local scope, and if it can't find it there, it will look in the global scope. So, if you want to use a variable that's defined in the global scope within your function, use the keyword 'global'.

****

Video 3: Nested Functions (4:06)

Let's now look at some more advanced functions you can write called nested functions. Nested functions are functions within functions. Recall that one of the reasons we write functions is we have some code that we want to run many times. Writing a function within another function serves the same purpose. You can now run a chunk of code several times within a function. Let's see how this works with an example. Here we have a function 'add_two' that adds '2' to a variable 'n'.

A regular function works well here. But what if you wanted to 'add_two' to multiple variables at the same time? We can do this with a regular function as well. All we need to do is add more parameters to the function header, and tell the function to 'add_two' to all of those parameters in the function body like this. However, you might notice a problem here. This function is really long, and in this example, we're only performing the calculation five times, but what if we were performing it a 1,000 times?

It would take us a long time to write this code. Plus, the more lines of code you need to write, the more likely you are to make an error. Because this function is performing the same process over and over again, it's a great candidate to be turned into a nested function. We can do this by writing an inner function that adds two to a variable. We also need to return the variable. This is something we haven't seen before.

The 'return' statement is similar to the 'print()', except 'print' is meant for a human user to see what a variable or other object contains, 'return' returns the variable so that it could be used by a function. Now, the values returned by this inner function can be used by the outer function which prints them out. Writing the function this way saves a lot of coding. You simply have to write 'a = a + 2' instead of each variable equals to itself plus 2.

Let's talk about the order in which this is processed. The inner function is processed first, then the outer function. You can actually write a function that has many levels, functions within functions within functions. The innermost will be processed first, then the next innermost, then the next innermost until you get to the outermost function. Let's try this out. Let's run the function 'add_two(1, 2, 3, 4,5)'.

First, the function will pass each of these variables into the inner function, add 2 to them and return them. Then each of those return values will be processed by the 'add_two' function which prints them out. So, we get '3, 4, 5, 6' and '7'. Recall that the names stored within a function are in the local scope. Once Python finishes searching within the functions inner to outer, Python will then search for names in the global scope.

To summarize, functions are useful when we want to run a process multiple times. Sometimes, within a function, we want to write a process multiple times, so we add a function within the function. This is called a nested function. With nested functions, the inner function is processed first, followed by the outer functions. Any names within these functions are in the local scope. So, the order of operations is, after searching through all the levels of a nested function, Python moves to searching the global scope as usual.

****

Video 4: Default and Flexible Arguments (4:32)

So far, the way we've been defining functions is with required arguments. But you can also define a function with an optional argument. For example, the function round, rounds the argument you pass to it to the nearest integer. So, '5.784' will round to '6'. However, the round function also has an optional argument of what place value you want the function rounded to. So if you write, 'round(5.784, 2)', you'll get '5.78', which is 5.784 rounded to two decimals. Why is it that the second argument is optional?

If the function has two parameters, how does it run even if we only specify one? This is what's called a default argument. In the function definition, the second argument is already specified. So, Python uses that information unless we specify otherwise. To understand this more clearly, let's look at some information about the function. We can do that by writing the function name followed by a question mark.

Here, we see the function 'Signature', which is the part of the function we defined in function header. We see the function name is 'round'. And it has two parameters, '(number and ndigits', 'number' is the number that will get rounded, and 'ndigits' is the number of digits after the decimal point that we want. Here it's specified that 'ndigits=None)'. So, unless you specify otherwise, the function will always assume that you won't have any digits after the decimal point.

We can do this with user-defined functions as well. Let's write a simple function called 'multiply' that multiplies two values and prints out the result. Here, when we call the function, we have to specify both parameters. Otherwise, we'll get an error. But let's say we want to make 'y' optional. And if it isn't specified, we'll just multiply x by two. We can specify this in the function header by typing 'y = 2'.

We can now use the function just like before where we specify x and y, or we can use it where we just specify x, and let the function run its default behavior where 'y = 2'. Another way you can add flexibility to your functions is by having a flexible number of arguments. To do this simply add an asterisk before your parameter. Then write a for loop in your function body that will go through each argument and run whatever process you specify.

Here I have a function that takes any number of names, then for each name, it will 'print('Hello, ' + name + " ! ")'. I can enter any number of arguments now. With one function, I can now say hello to Kyle, Mark, and Melinda. I can add as many names here as I want. To recap, many built-in functions have default arguments. That makes it optional for the user to enter those arguments, and if they don't, the function will run with a default in the function header.

Similarly, when we write user-defined functions, we can specify a default value for one or more of our parameters with an equal sign. That makes it optional to enter an argument for that parameter. We can either specify what the argument is when we run it or let the function rely on the default. We can also use an asterisk before our parameter name to indicate that any number of arguments can be passed to the function. We can combine this with the for loop in the function body so that the function applies the code to each argument entered.

****

Video 5: Handling Errors and Exceptions (5:17)

What happens when a function is used incorrectly? An error is thrown. You've already seen some of these errors like TypeErrors and ValueErrors. You may have also noticed that these error messages are very long and aren't always easy to understand. A short, clear error message can save you a lot of debugging time. In this lesson, we'll learn how to write functions that throw errors when they're used improperly, and you'll see how to write the error messages yourself in the function body.

Let's start with our function, 'hello_name'. This function concatenates 'hello' with the variable name. However, since 'hello' is a string, it can only concatenate to another string. Entering a non-string data type as the argument will throw an error. Here we see that entering '11', it throws a TypeError. There's a lot going on in this message, making it pretty difficult to understand. Let's rewrite this function to catch errors and print out an understandable error statement.

The code written to handle errors in a function goes in the function body using the keywords 'try' and 'except'. After the 'try' keyword, enter the code you want the function to try and run. After the 'except' keyword, enter the code you want to run if the code after 'try' throws an error. So, here we can include 'try' right after the 'docstring'. We follow with a ' : ', and make sure to indent the code after it four spaces. Then we add the 'except' keyword followed by a ' : '. Let's specify that if the function throws an error, we'll print("Make sure name is a string.

Put quotation marks around it.")' Let's now try to run the function incorrectly. We see a really clear error message, hence can quickly fix our mistake. Now, there might be other errors, other than TypeErrors that can occur in your function, but the way the except clause is written out, you'll print the same error message no matter what. To make sure that this error message is only thrown for TypeErrors, we can specify TypeError after the 'except' keyword like this.

Now, it will print out our message if the function throws a TypeError, but not if it throws any other kinds of error. For example, if I type in a variable 'a'. I get a NameError. I see the standard Python NameError here, rather than the error message I wrote. This ensures that our error messages will be specific to any potential errors. We can combine what we learned so far with conditional statements. This way, we can add more restrictions to what arguments can be entered into the function.

For example, let's say you want to make sure that names entered into the function are valid names. You define a valid name as one that is greater than one character. We can add an 'if' statement at the beginning of the function body. Here, I'm using the function len, which counts the number of characters in a string. Some saying if the number of characters in a string is '<=1' I wanted it to throw an error.

I followed this up with a colon, and on the next line, indented four spaces, I write, 'raise ValueError', and in the ValueError function, write the text that I want the error message to read. In this case, I'll say, '('name must be longer than 1 character.')'. Now, if someone tries entering 'L' as their name, they'll get a ValueError with our specific message. That's all for this video.

To review, we learned about the try, except structure to write clear, specific error messages. We also learned that we can add the specific error we want to catch exceptions for, right after the 'except' keyword. Finally, we learned how to add restrictions to our arguments using conditionals, and add specific errors when the conditions we specify are not met.

****

Video 6: Writing Lambda Functions (2:33)

By now, you're an expert at writing functions using the 'def' keyword. In this video, we'll learn another way to write functions using the 'lambda' keyword. This method for writing functions is quicker but has some limitations. The limitations are a bit out of the scope of this course, but as a general practice, you can keep in mind that lambda functions are typically used when you want to write a quick function that you'll only use a few times.

If you plan on writing a function that you'll use over and over again, stick to 'def' as we've been doing. Okay, let's take a look at how to write lambda functions. Here we have a function we defined using 'def'. It's called 'hello_name', and it concatenates 'Hello' to the argument. How can we write this function as a lambda function instead? We'll start with what we want to call the function. In this case, let's call it a 'hello_name_lambda'.

Then we follow it with an ' = ' After that, we write the keyword 'lambda', and the arguments the function will take. In this case, 'name'. Next, we write a ' : ' and, after the colon, what the function will do. Let's try it out. Notice that this is a much quicker way to write the same code, but we lose some of the readability. We also are limited in the code we can write in a lambda function. Let's try one more example.

Let's write a function that adds 2 to a variable. Again, we start with the function name. Let's call it 'add_two'. Then we write an ' = lambda', the function parameters and a ' : '. Now, we write the code that we want the function to run. We see it works as expected. Again, lambda functions are great for a quick function that you'll use briefly, but for functions you want to reuse multiple times, it's often better to stick to 'def'.

****

--END---