Module 3
Video Transcripts

Video 1: Functions and Arguments (5:49)

I'm now going to introduce you to functions in Python. A function is a piece of code that completes a certain task. Functions are reusable, so you don't have to keep writing the same code over and over. Python is great because it comes with so many built-in functions. This way, you don't have to write out the code for everything you want to do; it's already done for you. But if you want to custom function, you can write one on your own as well. For now, let's look at some common built-in functions.

Functions are followed by parentheses. What you enter in the parentheses are called arguments. Think of arguments as inputs to the function, then the function does some actions on that input, and produces an output. For example, a function you've already seen is 'print()'. The function 'print()' takes the argument of whatever you want it to print out, and prints it out. Simple enough, right?

Next, let's discuss the 'type()' function. 'type()' returns the datatype of the argument you put into the function. For example, here we see that 'False' is a boolean. Another function you should know is 'str()'. 'str()' creates a string out of an argument you enter into the function. Here, we entered the integer '4', and it outputs the string '4'. Next up, the 'del()' function. 'del()' takes a subset of a list as an argument and deletes the elements at that index.

To delete the third element from my list, I entered the list name followed by the index '2' into 'del'. Printing out the list now, we see that that item at that index was deleted. Now, let's look at some new built-in functions. 'max()', for example, takes numeric data and gives you the highest value. It can also take strings and tell you which one comes last in alphabetical order, 'min()' does the opposite of that; 'sum()' gives you the sum of numeric data, 'abs()' gives you the absolute value and 'round()' rounds the float that you enter to the nearest integer.

This is a good place to demonstrate how functions can have optional arguments. For this function to work, it needs at least the number that you wanted to round. But it can also take a second optional argument the number of digits you wish to round to the number 2. If I enter '1.623' into the function, it rounds to '2'. But if I specify that I want it rounded to '2' digits after the decimal point, I get '1.62'. Notice that arguments are separated by commas. The second argument is optional, because the function will still produce an output without it. Let's look at some more functions.

Let's try the function 'map()'; 'map()' takes two arguments; the first is the name of another function, and the second is the set of data you wish to apply the function to. For example, if I want to find the absolute value of each item in list 'x', I can use the 'map()' function. The first argument is 'abs()', the name of the function that returns the absolute value. And the second argument is the set of data I want to apply this function on, so 'x'. I'm going to assign this to the variable results, so we can see the outcome later.

Now, when I print the result as a list, I can easily see the absolute value of each item in list 'x'. This way, I don't have to run the 'abs()' functions separately on each element in the list. Let's check out the 'zip()' function. The 'zip()' function glues together elements from things

like lists or tuples. Here, I have a list of colleges my cousin applied to, and whether or not he was accepted. I want to create a new variable where I glue together each college, and it's acceptance status into a tuple, and create a list of those tuples.

Let's print it out to see. Now, we can see a nice clean list of tuples, each tuple containing the college name and the acceptance status. That's much neater than having to compare two separate lists. Okay, let's pause here. I bet you're wondering, "How will I ever remember all these functions? What other functions does Python have built-in? How do I know which arguments go into which function?"

Well, don't worry. You don't need to memorize all these things, because everything is spelled out in Python documentation. In the documentation, you'll see a bunch of functions. If you click on them, you'll see the arguments they take and what they do. You can also see the code that's being run when you run the function, so you know exactly what the function is doing. Right now, this code might not make much sense to you.

But it will after we learn how to write our own functions. Since Python is open-source, many people have written and shared their own functions, which are now easily available. Most likely, if you can think of a task that can be automated, a function has already been written for it, and you won't need to write one yourself. So, it's always good to check first before writing your own code, so you can save some time.

****

Video 2: Methods (3:54)

In this video, we'll learn about a special type of function called a 'method'. We'll learn the syntax for calling a 'method' and check out a few common examples. A 'method' is similar to a regular function in that it will take some input and produce an output. However, 'methods' belong to specific types. Let's look at some examples to clarify. A 'method' is written after the data you want it to work on. It's written with a dot followed by the method name followed by parentheses.

Here, we'll try out some 'string methods' on the variable name. The method 'upper' puts a string in all uppercase letters, running 'name.upper()' outputs 'EMMA WATSON' in all capital letters. If we want to just capitalize the first letter, we'll use the 'capitalize' method by typing 'name.capitalize()'. Another string method is 'replace'. While in the examples of 'upper' and 'capitalize', we didn't specify any arguments. If we do the same thing with 'replace', we get an error.

The error says that the method expected two arguments but got zero. The two arguments that 'replace' requires is the section of the string you want to replace and what you want to replace it with. Let's say we want 'Emma Watson' to actually be 'Emma Roberts'. Here's how we do it. In the parentheses after 'name.replace()', we enter the string we want to replace as the first argument and the string we want it, we want to replace it with as the second argument.

Okay, let's look at one more 'string method', 'find'; 'find' requires one argument; the string you want to search for within your string. Let's find the letter 'm'. The output says '1' because the first time you land on an 'm' is at index '1'. You can also search for strings with more than one letter like 'Watson', and it will tell you where that string begins, in this case, at

index '5'. If you enter a string that doesn't exist in your variable, you'll get an output of negative 1. Other data types, such as 'floats', have their own methods.

For example, '.is_integer' will output a boolean telling you if something is an integer. Here we get a 'False' because 'x' is a 'float'. Let's practice with some 'list' methods; 'count' will output the index position of the element in the list you're interested in; 'dec_temperatures_mood.count(20)' will tell you the index of the integer '20', in this case, '1'; 'append' will add data to a list. Here we add data for '"Dec 16"'.

This is an alternative to using the plus operator to add data to a list. Other data types have their own methods and just like with any other functions, Python documentation will tell you which functions are available, what arguments they take, if any, and what they do, so you don't have to memorize all the built-in Python methods.

****

Video 3: Writing User Defined Functions Part 1 (3:04)

In this video, we'll look at how to write custom functions. Functions that you write on your own are called user defined functions. These can be used to answer specific questions that are specific to your data and like all functions, they'll save your time since you can run many operations at once without having to rewrite the code each time. The syntax for writing your own function is comprised of two parts; the function header and the function body. The function header names the function and specifies which parameters, if any, it takes.

It starts with the keyword 'def' followed by the name you want to give your function. Remember, functions are always followed by parentheses and with a colon. The function body is the code that explains what your function does. This goes under the function header and is typically indented. Let's try this out in a Jupyter Notebook. Let's write a function that concatenates two strings and then prints the output.

First, we'll start with the function header. We always start with the keyword 'def', then the function name. Let's call this one 'hello_world', we always follow functions with parentheses and end the function header with a colon. When you hit enter on the keyboard, you'll see that it automatically indents for you. Now we can start writing the function body. This is what we want the function to do. I want this function to concatenate the string ' "Hello, + "World!" ' I'm going to assign this to a variable called 'x', and then I'm going to 'print (x)' out.

When we run the code, we notice that there's no output, but the function has now been defined. Now the function lives in your environment and you can call it any time. Let's call the function. If we type 'hello_world( )' and run the code, we see it prints out Hello, World! The function works correctly. While this is a very simple function, you can use it to understand the impact functions can have. Imagine writing dozens of lines of code that you have to use repeatedly.

If you write this code as a function, you can simply call the function each time rather than having to write the code from scratch. Let's look at our function one more time. It's good practice to include what are called 'doc' strings after the function header. This is a string that explains what your function does so that others using it can quickly understand what the function will do without having to read through the code. Simply use triple quotation marks and write a brief description.

****

Video 4: Writing User Defined Functions Part 2 (4:27)

You're now familiar with the basic structure of writing a function. But functions can also be more flexible. While the functions defined here can only print out 'Hello, World!', you can just as easily write a function that will print out hello followed by any name you'd like. To do this we start out the same way, 'def', the name you want for your function, parentheses, and a colon. However, before we move to the function body, we need to write a name for our parameter.

A parameter is a variable you'll specify when calling the function. You'll do this by entering an argument into the function when you call it. This makes your function more flexible, because you can have an infinite number of arguments you can use. Let's see this in action. Let's say you want to create a template that says hello to someone. Start with 'def', the name you want for your function, 'hello_name( ):'.

In the parentheses, we'll write 'name'. This is the parameter we need to specify when running the function. Now, let's write the function body. We want to concatenate the string, '"Hello, " + name', then we want to print out the result. Notice that name is not in quotations because it's a variable. Now, let's try running the function. If we run it without anything in parentheses, we'll get an error.

This error says that we're missing a required argument for the parameter name. Let's try again. We want to say hello to our friend 'Kate', so we'll enter '('Kate')' of the function. And we get the output 'Hello, Kate'. Imagine how useful something like this would be for email templates. You can automate them and save a lot of time. This could also be useful if you're working as a data scientist for a company that uses specific metrics as their key performance indicators or KPIs.

You can write a function that will quickly calculate these metrics each month. Given that month's data. Let's practice some more. Functions can also have multiple parameters. Let's write a function that greets someone. You want the function to be able to use any greeting, not just hello, and any name. So, we'll write two parameters here, 'greeting' and 'name' separated by a comma. 'x' is going to be the concatenation of 'greeting' and 'name'.

Then we 'print (x)'. Let's use this function to say good morning to George. That's pretty nifty. But what if you forgot the order in which you wrote your parameters? Does name come first or greeting? When calling the function, you can actually call the arguments in any order if you say what parameter it corresponds to using the equal sign.

So, you can say 'name = "Kelly"," and 'greeting= "Hello, "' and It will still print out 'Hello, Kelly' in the correct order. To end the video, let's discuss what this function actually does. If you look at the function body, it really just adds two variables. So, if we specify strings when we call it, it will concatenate them. But if we specify integers like '1' and '2', the function will just add them. We might want to explain that in the doc strings.

****

Video 5: Conditionals If Statements (5:18)

Sometimes you want your function to do one thing under one condition and another thing under another condition. We can specify this in the function body using the keywords 'If', 'Else', and 'Elif'. Let me show you how. Let's define a function called 'score_description'. That

takes the parameter '(score)'. It will print out a statement that tells you if you passed the test where a score of 65 is considered passing.

To do this, we start the function body with the 'if' keyword. Then we specify the condition. In this case, if the score is greater than 65. After the condition, we type a colon and go to the next line. On the next line, you can specify what will happen if the condition is true. Here, I'd like to 'print' out the message '('You passed the test!')'. Let's try it. If you got an 80 on the test, it will tell you that you passed.

But what if you forgot to study and ended up with a 40? Nothing happens. The function doesn't print anything out. If you want the function to print out a statement saying you failed the test, you can specify that with the 'else' keyword. Just type 'else:', and on the next line, write the code you want your function to execute if your condition is not true. Now, if we test this with 80, it will print out you passed the test, and if we test it with 40, it will print out you failed the test. You can also be more granular.

Let's say you want to categorize your scores into A, B, C, and so on, instead of just pass or fail. You can use the 'elif' keyword. Let's define a function letter grade that takes the parameter '(score)', using the 'if' keyword, we specify our first condition. If the 'score <= 100 and score > 93:', we'll 'print ("A")'. Moving on to our next condition, we'll use the 'elif' keyword. If the 'score <= 93 and score > 83:', we'll 'print ("B")'. Otherwise, we'll 'print ("F")'. Let's test this with a few scores to make sure we did everything correctly.

A '98' gives us A. An '88' gives us B, and a '60' gives us F. We can add more layers to this with more 'elifs'. For example, we can specify grade criteria for a 'C' like this. Something to keep in mind is that your 'if' and 'elif' statements will be read in the order that you write them. So, for example, here we have a variable 'x = 9'. Under it, we have conditional statements 'if x == 9:', we'll add '1' to it. If 'x < 10', we'll add '2' to it.

Otherwise, you'll add 3 to it. Notice that 'x' fits two of these conditions. It both equals to 9 and is less than 10. So, what will happen? We get a '10', meaning we added 1 to 9. The first statement that's true is the one that will be executed. If they were equal to '9.5', though, then the function would execute the second line. A final thing to note with 'if', 'elif' and 'else' is that indenting matters. The code under each statement needs to be indented four spaces.

In Jupyter Notebooks, this happens automatically. But if you're coding in another Python shell, you might need to do this yourself. Also, if you accidentally delete these spaces, your code will return an error. You can put these spaces back by hitting the Tab key or by hitting the spacebar four times. In this video, we saw just one example of using 'if', 'elif', and 'else'. But there are an infinite number of applications.

You can use this to send customers specific discounts based on whether they spent a certain amount of money, or to show targeted ads to specific demographics. Can you think of some other real-world applications?

****

Video 6: Conditionals: While Loops (3:55)

In addition to using 'if', 'elif', and 'else' to make our code do different things under different conditions, we can also set conditions using a 'while' loop. Recall that, when you write code using the keywords 'if', 'elif', and 'else', your code will run through each statement in the

order that you write it. And, once it hits a condition that's true, it will do what your code specifies and end. That means your code only runs once. Instead of 'if', you can use the keyword 'while'.

Rather than just checking the condition once, it will keep checking it and as long as the condition is true it will keep running; 'if' is used if your codes doesn't change your data; 'while' is used if your code does change your data. Let's look at an example. We'll work with the variable 'x' which right now is equal to 2. A 'while' loop is written just like an 'if' statement. The keyword is followed by the condition and a colon.

After the colon we write the code that we want to execute as long as the condition is true. We'll say that while 'x < 1000', we're going to multiply it by 10. If we run the code, we see the output 20, 200 and 2000, then it stops. What the code did was multiply 2 by 10 which turned 'x' into 20, then it multiplied that by 10 and got 200, then it multiplied that by 10 and got 2000. Now, 'x < 1000', so the code stops running. To solidify your understanding, let's try this one more time with 'y = 15'.

The code stops when 'y' is equal to 1500 because it is now no longer less than 1000. Notice that, in order to change the variable, we have to specify that we are reassigning the variable to the new value that we calculated. If we remove this part, what will happen? The code will keep running and printing out 15 because, while the code is multiplying 'y*10', it's not actually changing the value of the variable 'y'. So, it keeps printing out the value we assigned it, 15.

That also means that 'y' will never be greater than 1000, so the code will run forever. You can stop the infinite loop by clicking the stop button up here. Just like an 'if' statement, you can use a 'while' statement in a function. Here, we define a function 'subtract_one' that takes a parameter 'x'. While 'x > 2', the code will subtract 1 and 'print (x)'. Let's try this function out with the argument 5. We see it works as expected.

We can also combine the 'while' keyword with the 'else' keyword to have our code do something once our 'while' condition is no longer true. Here's the same function except now I added the 'else' keyword. It will 'print' out ('X is no longer greater than 2!')' once the 'while' condition stops being true. Let's try this again with the argument 5. The same thing happens except once X is no longer greater than 2, we get the string from the else statement.

****

Video 7: For Loops (5:33)

In this video, we will learn how to use a 'for loop'. To understand how a 'for loop' works, we first need to discuss the kinds of objects it works on. 'For loops' work on iterable objects. An iterable object is an object that returns its numbers one at a time. There are many types of iterable objects, but the ones we've seen so far are lists, tuples, and strings. A list or a tuple can return its elements one by one, and a string can return each character one by one. Let's see how we can use 'for loops' with these objects.

Here, we have a list called 'test_scores', with the elements '[40, 65, 91, 92,' and '93]'. The professor decided to curve the scores and add seven points to each score. We can do this using indexing. We'll index each element individually and add '7' to it. That's one line of code for every element. Imagine we had a whole class of 100 students here. Or what if we have a data set of 10,000 data points? Even with copy-paste, we'd be here for a while.

It's much more efficient to use a 'for loop'. To write a 'for loop', start with the keyword 'for', then a variable to call each element. You can call this whatever you want, but it's good to be descriptive. Since the elements here are scores, I'll use the variable 'score'. Then we write the keyword 'in', and then the iterable, in this case, the list 'test_ scores'. End this with a ':' and go to the next line. On the next line, you'll write the code you want to run on each element.

In this case, because I want to add '7' to each element, I'll reassign the value of 'score' to 'score + 7'. That's much easier than adding '7' to each element individually. We can do this with any iterable object. Let's try it with a string. Just for fun, let's print out each letter of the string "LOL" separately, and catenate it with an exclamation point. Let's go back to lists for a few minutes though.

Here we have a list called 'my_list' with the elements '[1,2,3,4,' and '5]'. If we want to multiply each element by '2', it's pretty easy to do that with a 'for loop'. But let's say I don't have a list to define, and I want to do this for all integers from 1 to 100. Rather than listing all the numbers, I can use the 'range()' function. The 'range()' function takes two arguments. The first is where you want the numbers to start, and the last is where you want the numbers to end.

To get from 1 to 100, you'll need to stop at 101, because the last number is not included. We can keep the body the same. Here's my really long output of numbers calculated really quickly. A lot of the concepts we've learned so far can be combined. For example, a 'for loop' can be used with a conditional statement. Let's say you want to add '7' to each score, but only if the score is below 80. How would we do this? We'll start with the list of test scores. We'll use the 'for loop' to iterate over the list, and we'll follow up with an 'if' statement.

We'll add '7' to test scores that match the criteria. Then we'll print out each score. See here that '7' was only added to the first two scores. If the professor plans on doing this for every test, rather than writing the code each time, he can write it as a function. Here, we create a function 'score_curve' that has the parameter '(test_scores)'. The function body will be the same as the code we previously wrote.

Now, we can just enter the scores into the function, rather than writing the code from scratch each time. To summarize, we learned that 'for loops' can efficiently execute some code on all elements of an iterable object at once. They can also be used with conditional statements to selectively apply the code to specific elements, and in a function to save as a chunk of reusable code, saving you even more time down the road.

****

Video 8: Looping Through a Dictionary (1:46)

By now you should be familiar with looping through a list. Now, let's go over how to loop through a dictionary. As a reminder, dictionaries are created in curly brackets and are made up of key-value pairs. Here, we have 'test_score' data in a list. Applying a loop to the list looks like this. Now, here's the same 'test_score' data but in a dictionary. Each key is a student's name, and each value is the score they received.

Since we want to loop through both the key and the value in the dictionary, we may expect our loop to look like this. However, we get an error because Python doesn't automatically

loop through both keys and values. Fortunately, there's a simple workaround for this, the '.items() method'.

The '.items() method' creates iterations of key-value pairs so you can loop through them. Let's run the same loop with the '.items() method'. It works. One thing to note about the syntax is that, no matter what you call your keys and your values in your loop, in your syntax, the name for the key must come first; and the name for the value must come second.

Also, keep in mind that dictionaries are unordered, so the loop won't necessarily go through the dictionary in the same order you have it typed up. To sum up, looping through a dictionary is the same as looping through a list except you need to use the '.items() method' and make sure your key variable comes before your value variable in the loop.

****

----------**END**----------