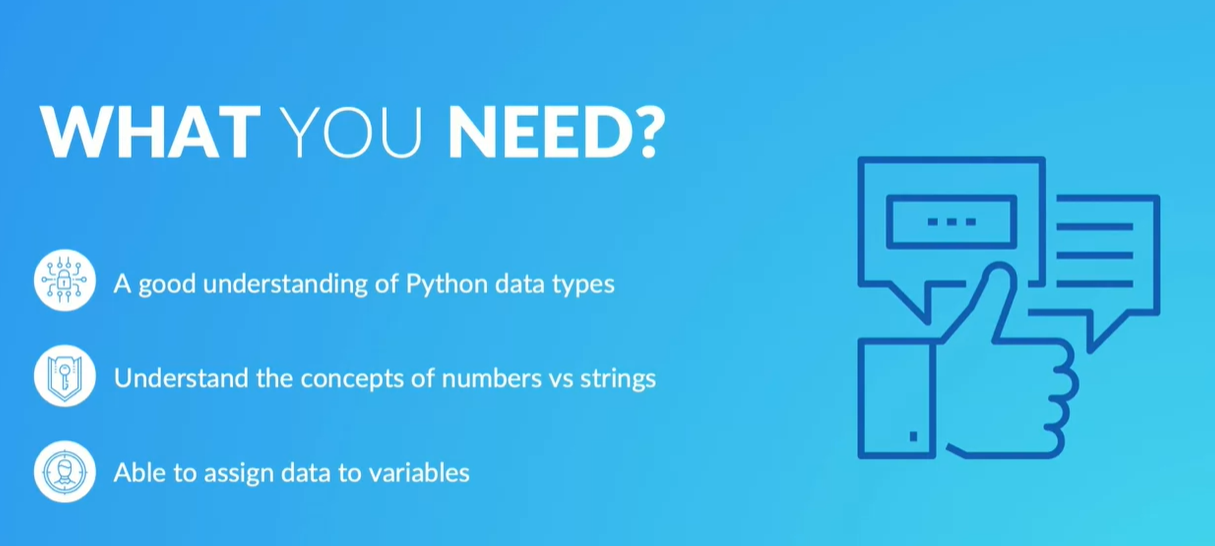
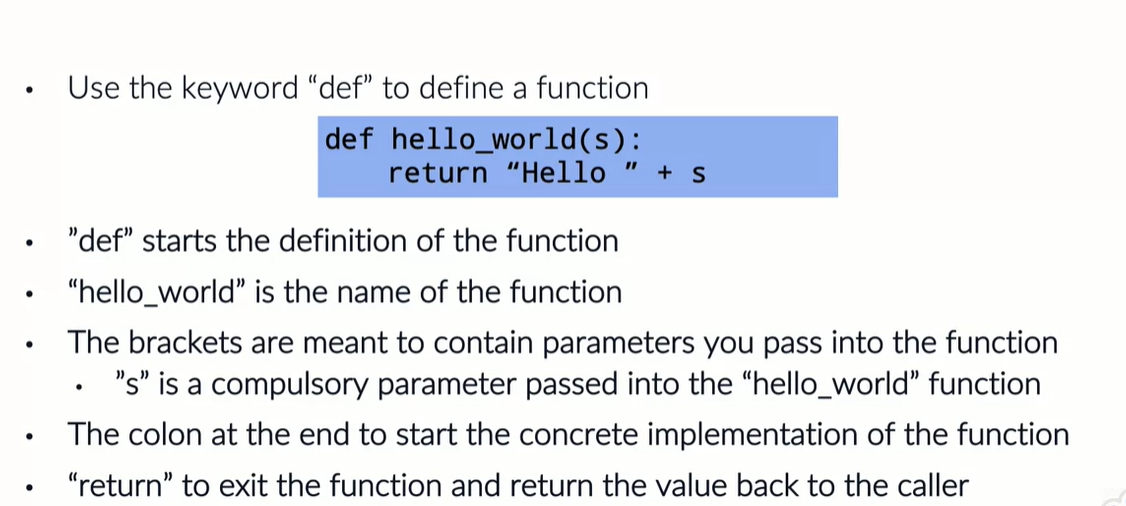
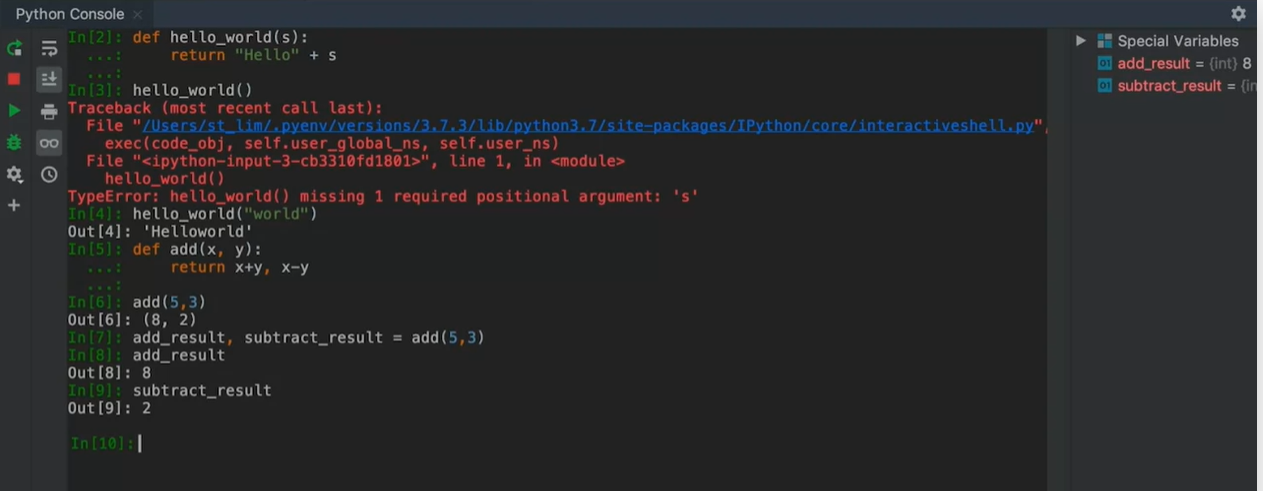
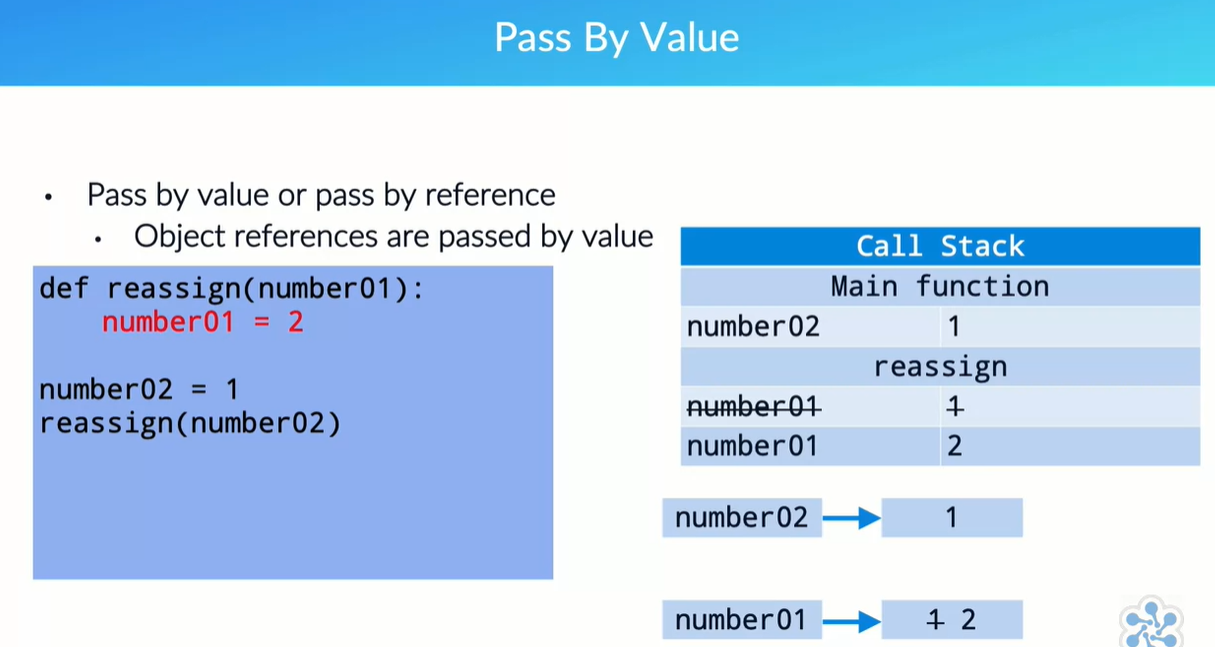
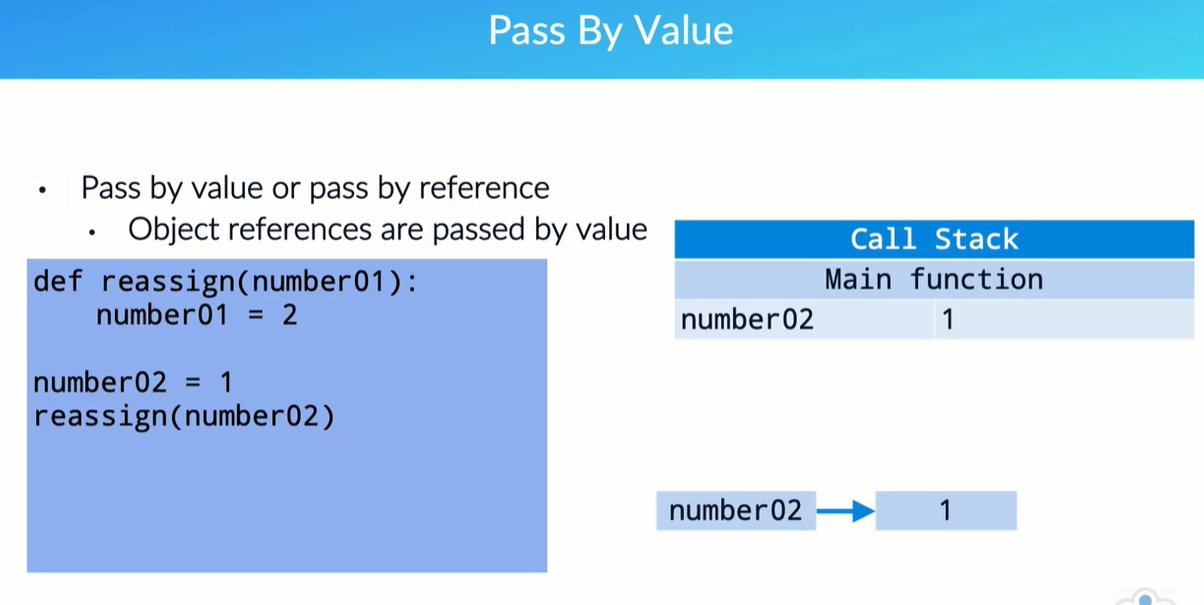
Functions



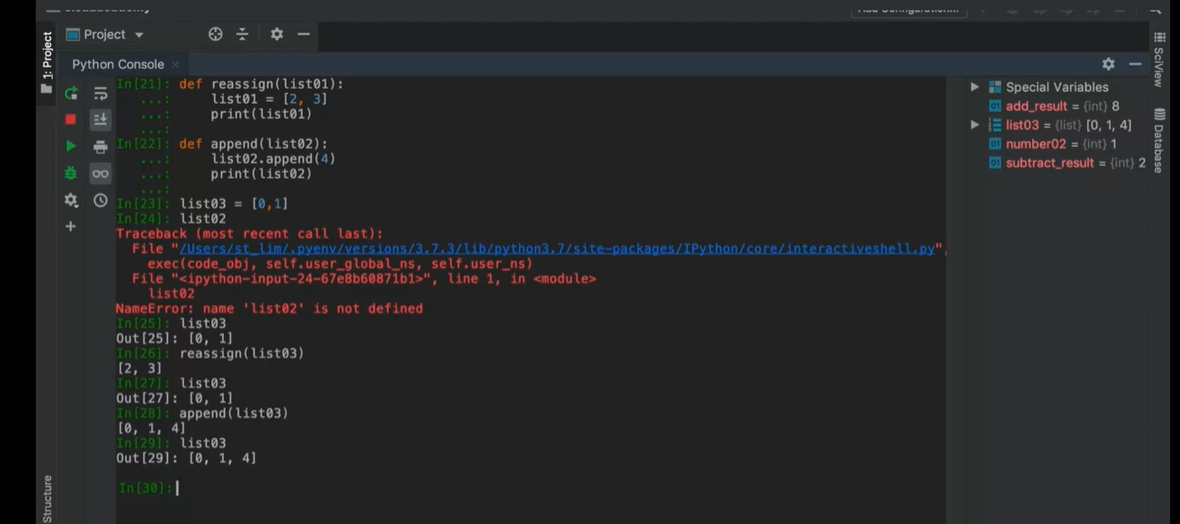








once the function exits



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Transcript:

Okay, welcome back. In this lecture, I'm going to introduce you to the concept of Python functions. Functions are an essential part of Python programming. Understanding how when to use them is an extremely important skill that needs to be mastered. So in this lecture, we're going to introduce you to a function and provide several demonstrations of developing with functions. In order for you to start, you need to have a comprehensive understanding of the Python data types we have previously explored, you need to understand the general concepts of coding with numbers and strings, and know how to assign data to variables and how to manipulate and use variables. Okay, to start with, functions are created by using the keyword def, D-E-F, which is used to start and define a function. This is put at the front of a line to define a function that you want to later use elsewhere in your code. Taking the example of our original hello\_world application, we start off with this particular snippet of code. In this particular snippet of code, we have a def keyword that starts the line. Now this basically tells Python that you created the definition of a function. The function name that you are trying to define here will be hello\_world. The brackets are meant to contain parameters that you can pass in to the function. You can pass in any number of parameters. Ideally, we wouldn't want to pass in too many. Usually, you need to know what your function needs to be doing and based on that, you pass on the parameters you need in order to operate on it. In this case, we are using the s parameter to pass in a compulsory parameter into the hello\_world function. This means that no matter what you do, you cannot invoke the hello\_world application without that required parameter. The colon at the end of the first line that starts with D-E-F is used to indicate the start of the function's implementation. The function is also used typically to indicate the start of a particular section of the program. Now, in this particular program, we used the return to exit the function and return a value back to the caller. In this case, you can see this here as it is returning Hello with the input parameter s appended. Now, it is very important that you note that the return function is prepended with a blank space. This is extremely important in Python, which respects blank space in order to perform indentation, and also to identify that this section of the code belongs to the function that you defined earlier. If you have multiple lines and one of the lines does not have the indentation, Python will recognize this as a requirement that you plan to resolve further up the function chain. So, without much further ado, let's go into a terminal to understand how this works. So in the console, you can see we are going to start typing the hello application originally defined. This has auto-indented, so do be mindful about the auto-indentation when you run your own program. You'll recall that I said that the s is a compulsory parameter. So, if you do not present an argument there or even pass in a parameter, this will be the error you will get. Passing in a variable here, we'll try a simple world here. Notice how there is no space between the word Hello and world. This is because the string has directly appended.

Now, consider the case where we want to return multiple values. For example, if I have an addition function where I add x to y, I can return both x plus y along with x minus y, and this is still a valid function. So let's test this. In this case, we call the add function with five comma three, and we should get an output in this form. How many of you can recognize that this is actually a tuple? We are now able to assign the multiple return values to multiple variables like this. I will then be able to find that my add result is equal to eight and my subtract result is equal to two, and the list goes on, as in you can specify as many returns as is necessary. Let's now stick back to the slide so that we can understand how data is passed into the application. One of the key problems when understanding Python, or in fact any programming languages, is the question of how functions accept parameters using the concept of pass by value or pass by reference. Python passes object references by value. Let's take a closer look and understand what this means in code and physical memory perspective. So, let's take for example this particular piece of code where I define a function reassign. The defined function takes an input parameter, number01. Within the function, it assigns the value 2 to number01. Back below in the main part of the program, as you can see, 'cause it is not indented and sits in the same indentation as the def or the function, I have a number02 variable which is assigned the value 1. And then I invoke the function reassign with this variable passed in as the parameter. Now, pay attention because we are going to walk through this call stack. In the call stack, when we initialized the program, we will start off with what we call the main function, this is the main body of the program. Now, you will have done the definition of the reassign function somewhere. Here, once the first line is invoked, number02 is assigned with the value of 1. What this means in memory is that I have something that is used to indicate the variable called number02. It is assigned to some memory allocation location where I will see the value 1. Now, when I invoke the function reassign, then I would have within it a parameter called number01. number01 will be initialized to be nothing, but because when I do the invocation of the function reassign, I passed in the value of number02. Now the value of number02 gets copied into the variable number01 within the reassign function. So, once it's copied there with no linkage between number02 and number01, take note again, there is no linkage between these two, next, an assignment takes place within the reassign function. number01 is assigned the value 2. What this means is that the local variable number01 is now getting the value 2. We have taken out the value 1 because we no longer look at it anymore and we have replaced it with the value 2. Within the context of the scope of the function reassign, the value of number01 is 2. Now, when I then exit from the reassign function because there's no more code to process after the assignment of value 2 to the variable number01, then the function would simply exit and the respective call stack would be cleared. The reassign function will no longer be in memory. Now, when I look at the variable number02, I should see the value 1, and this should be very clear to any user. Once you pass in the value, it is copied into the call stack for the function and exists within the scope of the function. And then when the function exits, the contents of the call stack and the data in scope for that function will be discarded. The resulting main call stack will still know that number02 still has the value 1.

Previously, we were talking about how you pass parameters into the reassign function, so let's now demonstrate this so that you can understand how it actually works. So here, you can see that we are executing the function reassign with the variable number02 passed into it. At this moment, number02 is equal to 1. When execution is returned from the reassign function, you will find that number02 is still equal to 1. Let's now update the reassign function so that we can see what's going on within it. We'll update the reassign function by adding in an extra print statement to print out the value of the location number02. As you can see, within the function reassign, the value assigned to the number01 variable is printed out as 2, but when we look at the variable number02, because the value was not passed back into the main call stack, number02 stills retains the value 1. So, understanding how this works when using built-in parameter types, such as the integer type, is fairly straightforward. The problem then is when you use reference type objects, such as lists. So for example, consider this code snippet, where we again used the reassign function, but this time, together with a new function named append. Then in my main program, I assign a value to the list03 variable, then pass this variable into the reassign function. And then later also pass it into the append function. Now, this is where things get a little bit tricky. Try and focus on the explanation given here so that you understand what's going on. We start with the call stack. The program starts with the variable list03 being assigned a new list containing the values 0 and 1. This would result in the memory space looking like this for the current state of the variable list03. Notice that list03 is not pointing directly at the values 0 and 1. This is because list03 is of type list, and therefore, has to maintain metadata about that list. That metadata could include things like the length of the list. The key point here is the fact that we need to have a location where we can say this is where the list starts. This is indicated by the box containing the asterisk that the list03 variable points at. Now, when the function reassign is called, a variable named list01 is initialized within the scope of the function. This creates a new memory object called list01 and is initialized as nothing. But because you passed in list03 earlier, we take a copy of the reference, not the list itself but the reference, and we will copy it into list01. So we now have the list01 variable, which exists within the scope of the reassign function pointing to the same array of numbers that list03 points at.

Now, when the reassignment takes place within the reassign function, what this does is change the reference pointer for list01 to point to a brand new list containing the values 2 and 3. Notice in memory that nothing has been removed or changed with respect to list03, only list01 has changed, and this happens within the scope of the reassign function. The reassign function now has list01 pointing to a list containing values 2 and 3. When the reassign function exits, does it still remember list01? Well, the answer is no, it doesn't, because outside of the scope of the application, there is no interest in list03 which has and still retains a pointer to a list containing the values 0 and 1. The application now invokes the append function and we create a new variable named list02 within the scope of the function. Again, we make a copy of the pointer into list02 and again, list02 will start off by pointing to the same list as list03. This is all held within the same memory function. Next, we invoke the append operation on list02, passing in the value 4. So, what is gonna happen? Does list02 then create a new copy? No. Instead, there are now two references pointing to the same list. No changes have been made to the underlying list. Instead, what has happened is that the append function after has been called, appending the value 4 to the same list that list03 points to. Again, since list03 is also pointing to the same location, list03 will also end up seeing the same mutation happening. It is because of this that when the append function exits with list02 being dumped out of scope, list03 will still point at the same memory address location and will still see the list with the values 0, 1, and 4. Okay, so let's demonstrate this with an example to ensure you understand what's going on 'cause it is a little bit complex. I'm also going to make use of the print function so that we can see clearly what values each variable holds at a particular point within the execution. So, in the main program, now we are going to say that list03 is equal to a new list containing the values 0 and 1. This is what list03 looks like. Notice I previously typed list02, which does not yet exist anywhere in memory. Again, list03 currently looks like this and is a list with the elements 0 and 1. Now, when I call the reassign function with list03 being passed in as a parameter, it is copied into the call stack as parameter list01. You will see that it is then reassigned with a list containing the values 2 and 3 within the call stack. But over here in the main function, list03 still retains its initial value. However, when we next call the append function, we will see that list02 internally has been printed and it has been appended with the value 4. So, list03 now also has the value 4 within it. Okay, hopefully that's quite clear. Try it out for yourself. So, that concludes our understanding of how Python passes object references by value. Now that we've gone through this lesson, you should have a better understanding of the concepts of functions and how Python passes object references by value. This is important, and it forms the basis of how we will interact with functions in future courses.