Functions in Python: Introduction

Explore how Python facilitates code reuse by using functions in this 17-video course, which shows learners how to define functions, learn passing arguments to functions, and returning values from functions. The functions you will examine change the state of the program, may have side effects, and have observable effects other than their return values. Since functions with side effects are hard to parallelize and use in a distributed environment, you will learn correct ways of returning values from functions. First, you will learn how to invoke functions by using both positional and keyword arguments. You will next work with positional input arguments in custom functions, and learn that these are required arguments, and how to order these arguments to invoke your function. You will next learn to use variable length arguments in defining custom functions. Finally, you will learn how keyword arguments or named arguments are a way to make the intent behind function invocation absolutely explicit, and help prevent bugs in programs that are especially hard to detect.

Table of Contents

Course Overview

[Video description begins] *Topic title: Course Overview* [Video description ends]

Hello there, welcome to this course. My name is Janani Ravi and I will be your instructor for this course.

[Video description begins] *Your host for this session is Janani Ravi. She is a software engineer and big data expert.* [Video description ends]

A little about myself first. I am the co-founder of Looneycorn, a studio for high quality video content. Before founding Looneycorn, I worked at various companies including Google for seven years in New York and Singapore, and Microsoft for three years in Redmond.

And before that, I attended grad school at Stanford. Python is fast emerging as the programming language of choice for data scientists and data engineers. And at the heart of Python's appeal is the manner in which it allows object-oriented and functional programming to be combined.

Code written in a functional programming style is particularly important in applications that involve big data and real-time processing. Because such code can be parallelized and shipped off for distributed execution relatively easily.

This course, Introduction to Functions in Python, you will see how exactly Python facilitates code reuse using functions. You will learn how to define functions, pass arguments into functions and extract return values from these functions.

Functions that change the state of the program, that is, have observable effects other than their return values, are said to have side effects. Functions with side effects are hard to parallelize and use in a distributed environment.

You will learn correct ways of returning values from functions, as well as of invoking functions using both positional and keyword arguments. Keyword arguments, also known as named arguments, are a great way of making the intent behind function invocation absolutely explicit.

They also help prevent especially hard to detect bugs in programs. By the end of this course, you will know why functions are a great code reuse and code composition mechanism. And you will also know how functions can be defined, implemented and correctly invoked in Python.

Getting Started with Functions

[Video description begins] *Topic title: Getting Started with Functions. Your host for this session is Janani Ravi.* [Video description ends]

Hopefully at this point in time, you're pretty comfortable writing simple programs in Python.

[Video description begins] *A Python IDE interface opens in a browser window. The URL in the Address bar is localhost: 8888/tree. A blank Jupyter notebook is open here. Below the page header bar, three tabs display, Files, Running, and Clusters. The File tab is active now. Below the tabs, in the right corner are three buttons, Upload, New, and Refresh. The New button is highlighted now.* [Video description ends]

We've written a lot of code, but we haven't reused our code in any effective way, and that's what we'll study here in this course.

[Video description begins] *The host clicks the New button and a drop-down appears, containing names of existing notebooks under a section titled Notebook. Presently, just one notebook, Python 3, displays here. Underneath, other types of files, such as Text File, Folder, and Terminal, display in a section titled Other.* [Video description ends]

Create a new Jupyter notebook here running the Python 3 kernel.

[Video description begins] *As she selects Python 3, an Untitled notebook opens. The text, Untitled, displaying in the notebook header bar is highlighted now. Below the file name, a Menu bar displays, containing menus such as File and Edit. Below the Ribbon, the first code cell displays. The command prompt reads as In [ ]: An empty input box displays after the command prompt.* [Video description ends]

I'm going to rename my notebook from the default untitled name to be called Functions. Because functions are what we are going to use in order to write reusable bits of code.

[Video description begins] *As she clicks Untitled, a dialog box titled Rename Notebook displays. It contains a field labeled Enter a new notebook name, where she types Functions. Buttons labeled Cancel and Rename display below.* [Video description ends]

Let's keep our code fairly simple to start off with before we move on to more complex stuff.

[Video description begins] *In the first code cell, she types the following three lines of code: line 1: print ("My name is John") line 2: print ("I live in Boston") line 3: print ("I am a data analyst and I love what I do!").* [Video description ends]

Let's say this is code which you write to introduce yourself. You say, my name is John. I live in Boston. I'm a data analyst and I love what I do. Now, the code that we've written here comprises of three print statements all of these are one of. So if you have to introduce yourself to someone else, you'll have to write all three print statements once again.

[Video description begins] *As she executes the code, the number 1 displays within the square brackets after the word In. The following lines of output display below the code: line 1: My name is John line 2: I live in Boston line 3: I am a data analyst and I love what I do!* [Video description ends]

Imagine that you're at a conference and you've met someone, you've just introduced yourself and you go ahead and meet someone else. Your introduction starts all over again. But instead of verbally having to introduce yourself, let's say you had a business card.

You might meet a bunch of people and hand out your business card and that would serve as your introduction. A business card is a grouping of everything that you need to introduce yourself. And that's exactly what a function is. A grouping of commands and statements that can be reused without having to write these commands and statements over and over again.

[Video description begins] *In code cell 2, she types the following lines of code: line 1: def my_introduction ( ) : line 2: print ("My name is John") line 3: print ("I live in Boston") line 4: print ("I am a data analyst and I love what I do!").* [Video description ends]

Observe that the code that I've written here has the same three print statements as earlier. My name is John, I live in Boston, I'm a data analyst, etc., etc. But I have grouped these print statements into a function called my_introduction.

How do I know it's a function? Well, it's preceded by the def keyword. The def keyword is what you use to define a function. If you want to define a function that is a reusable bit of code, you need to proceed the name of the function using the def keyword.

So we have the def keyword then my introduction is the name of the function. Observe that it's followed by brackets or parentheses. Every function will have the function name and then parentheses with a few exceptions that we'll encounter later on.

After the parentheses, we have the colon indicating that the function block is about to begin right after that. Once you have the colon, you write all of the statements that you want executed when you invoke or call this function.

And this group of statements makes up your reusable bit of code. Once you hit Shift+Enter on this bit of code, your function has been defined and it's ready to be used. Now, defining a function only means that you've told Python that this is the function.

And these are the statements or commands to execute when this function is called. It doesn't actually run the code associated with the function until you invoke the function. And really we are already familiar with what it means to invoke a function. We use the name of the function and use parentheses, and this is a function invocation.

[Video description begins] *As she executes the code in code cell 2, the number 2 displays within the square brackets after the word In. In code cell 3, she types the following code: my_introduction ( ).* [Video description ends]

So we are now invoking not a built-in function in Python as we have so far. But we are invoking our own custom function, my_introduction. And this function invocation will simply execute all of the code that lies within that function.

[Video description begins] *As she executes the code in code cell 3, the number 3 displays within the square brackets after the word In. The following lines of output display below the code: line 1: My name is John line 2: I live in Boston line 3: I am a data analyst and I love what I do!*. [Video description ends]

The invocation of the my_introduction function executed all of the statements that were within the function block. Now, it so happens that we had just the three print statements. Those three print statements were executed.

And the introduction has been printed out to stream. Observe that instead of writing the three print statements again for the execution, I simply invoked the function.

The function was a reusable bit of code. Now let's say I invoke the function again, my_introduction followed by parentheses. The same statements within the function will be executed once again and their results displayed to stream.

[Video description begins] *In code cell 4, she types and executes the same code as in cell 3. The output is also the same*. [Video description ends]

Observe how our custom function is like a business card. You simply invoke the custom function that's like handing over your business card. And all of your introductory details are available to the receiver.

Similarly, you don't have to write out the three print statements each time. You simply invoke the function and your introduction is complete. Now that we know what a function is, let's define another function. And I'm going to call this hello_world.

[Video description begins] *In code cell 5, she types the following lines of code: line 1: def hello_world ( ): line 2: print ("Welcome to Python!") line 3: print ("This is outside the function")*. [Video description ends]

Notice that our hello_world function name is preceded by the def keyword. Then we have the function name, the parentheses, then the colon. Observe that statements within the function block are indented,

[Video description begins] *An arrow points to line 2 of the code*. [Video description ends]

just like statements within the if block or within a for or while block. The only statement here within the hello_world function is the print statement Welcome to Python.

Observed that there is another print statement in the same code cell. But that print statement is not indented. When it's not indented, and it's aligned with the def keyword, it basically means that that statement is outside of the function code block.

[Video description begins] *An arrow points from def in line 1 to print in line 3*. [Video description ends]

So we have a print statement, This is outside the function. Which is outside of the function code block and a print statement within the function code block. Now we can prove this by executing this code which will define the function.

And then execute the print statement, which is outside of the function block. Observe that the code that is within the function block is not executed.

[Video description begins] *The following output displays below the code: This is outside the function. The number 5 now displays after the word In in the command prompt of code cell 5*. [Video description ends]

The print statement Welcome to Python is a part of the function. The function has only been defined, but not invoked, which is why we don't see Welcome to Python onscreen. Let's go ahead and invoke the hello_world function.

[Video description begins] *In code cell 6, she types the following code: hello_world ( ).* [Video description ends]

That is the hello_world name, followed by the parentheses, that is a function invocation. And that will print out to screen Welcome to Python.

[Video description begins] *The following output displays below the code: Welcome to Python!. The number 6 now displays after the word In in the command prompt of code cell 6.* [Video description ends]

Observe that when we invoke this function, only Welcome to Python was printed to screen. The print statement, This is outside the function, was not executed and printed. This is because that print statement is not part of the function code block.

Only the statements and commands within the function code block will be executed when you invoke a function. The functions that we're working with here in this demo are custom functions. Custom functions have to be defined before they can be used or invoked.

[Video description begins] *In code cell 7, she types the following code: my_new_function ( ).* [Video description ends]

Let's say now you try to invoke a function called my_new_function. This is not a built-in function in Python. Python doesn't recognize what it is. It has no idea what you're talking about when you invoke my_new_function without first defining it.

When you hit Shift+Enter and try to execute this my_new_function, you'll find that Python very clearly tells you this is a name error. The name my_new_function has not been defined. It has no idea what you're trying to do when you invoke this function. In order for this to work, you first need to define your function before you invoke it.

[Video description begins] *In code cell 8, she types the following lines of code: line 1: def my_new_function ( ): line 2: print ("This is now defined") line 3: my_new_function ( ).* [Video description ends]

Observe that we have multiple lines of code within this code cell. The first is the function definition. We've defined my_new_function, which simply prints out to screen, This is now defined.

Once we've defined it, we can invoke it. And the second command here is my_new_function followed by parentheses, that is the function invocation. And this will now work. Observe that the code within our my_new_function function is executed. And you see printed out to screen, This is now defined.

[Video description begins] *The following output displays below the code: This is now defined.* [Video description ends]

Your custom function is available for invocation only after you have defined it and this is true in all situations.

Working with Functions

[Video description begins] *Topic title: Working with Functions. Your host for this session is Janani Ravi.* [Video description ends]

Now let's talk about the importance of indenting your code. Here I've defined a function called will_not_work_function,

[Video description begins] *The Functions notebook from the previous demonstration opens. Its first 8 code cells are populated with code. In code cell 9, the host types the following lines of code: line 1: def will_not_work_function ( ): line 2: print ("This indentation is all wrong").* [Video description ends]

I've named it such because this won't work. And the reason for that here is because I have an error in how I've defined this function. After the colon, all of the statements that belong to the function which are part of the function block should be indented, and the print statement that I have here isn't indented.

Which means it's not a part of the function block, your function definition is syntactically wrong. Python sees that there is a colon at the end of your defined statement.

[Video description begins] *As she executes the code, an IndentationError displays. It says the following: expected an indentation block.* [Video description ends]

It's expecting an indented block after that, you did not provide an indented block, and which is why you see this indentation error. When you want code to be reusable as a part of your function, make sure you indent it correctly within your function defined statement.

Because you're just getting started with functions in Python, the functions that we're writing here are all very simple. However, in the real world, you'll find that functions are often complex, with conditional checks, for loops, while loops, all of the bells and whistles. Here is a function that will work because our indentation is correct.

[Video description begins] *In code cell 10, she types the following lines of code: line 1: def will_work_function ( ): line 2: print ("This indentation is all right!") line 3: if 10>5: line 4: print ("Well 10 is greater than 5").* [Video description ends]

I've called it will_work_function. After the colon in the function definition, we have the right indentation. We have a print statement which says this indentation is all right, and then we have an if conditional check within this function.

This if basically checks to see whether 10 is greater than 5. You know that it is, and then it'll print out to screen well, 10 is greater than 5. Observe that when you have an if condition within a function, the statements within the if block have to be further indented.

The second print statement here, well, 10 is greater than 5, is part of the if conditional block. Which is why there is a further indentation indicating as part of the function but also part of the if block within this function.

With this function defined, let's go ahead and invoke this will_work_function. And it'll execute both of the print statements because our if condition was true.

[Video description begins] *In code cell 11, he types the following code: def will_work_function ( ). The following lines of output display below the code: line 1: This indentation is all right! line 2: Well 10 is greater than 5.* [Video description ends]

This indentation is all right, well 10 is greater than 5. If it so happens that the if conditional check within the function evaluated to false, then the second print statement would not have been executed. Remember, conditions, for loops, while loops all work in exactly the same way when used within a function.

Here is a slightly more complicated function which shows you how you can include for loops within a function as well.

[Video description begins] *In code cell 12, she types the following lines of code: line 1: def a_more_complicated_function ( ): line 2: for i in range (10): line 3: print ("i is now:", i).* [Video description ends]

I have the function definition, and then I have an indented block which contains a for loop. For i in range then i is now and I print i out to screen. Let's define this function and go ahead and invoke it using the name of the function. Observe something interesting here.

Once you've defined your function, you can see that your custom function name becomes a part of the autocomplete suggestions offered in Jupyter Notebook.

I've simply typed in a here, and this gives me a range of options for autocomplete, and the first option here is a more complicated function, the custom function that we just defined. I'm going to select this option and invoke this function, and this will execute the for loop that we had set up within this more complicated function.

You can see that on function invocation the for loop is executed exactly as though you typed in this for loop within your Jupyter notebook code cell directly. i ranges from 0 all the way through 9, and all of the steps are printed out to screen.

[Video description begins] *The output displays ten lines. Line 1 is i is now: 0. In each successive line, the number after the colon is incremented by 1 till it reaches 9*. [Video description ends]

Functions are typically used to execute something perform actions. So make sure that you name your functions meaningfully. Functions can include underscores, capital letters, as well as lowercase letters.

[Video description begins] *In code cell 14, she types the following lines of code: line 1: def _Functions_Can_Be_NamedLikeThis_123 ( ): line 2: print ('This works!') line 3: _Functions_Can_Be_NamedLikeThis_123 ( )*. [Video description ends]

You can also include digits within functions provided that your function name doesn't start with the digit. Observe this function here. It starts with an underscore, uppercase, lowercase, and digits are all included.

This is a completely valid function, and can be executed. On the other hand, if the name of your function starts with a digit, this is not valid in Python.

[Video description begins] *The following output displays below: This works!*. [Video description ends]

So make sure that you don't have your function names begin with a digit. As you'll see here on screen, you'll see that the definition of this function will give you a syntax error. This is invalid syntax as far as Python is concerned.

[Video description begins] *In code cell 15, she types the following lines of code: line 1: def 123this_does_not_work( ): line 2: print ('This does not works'). The output of this code is SyntaxError: invalid syntax*. [Video description ends]

Functions as Objects

No transcript available

Input Arguments - Invoking Functions

[Video description begins] *Topic title: Input Arguments- Invoking Functions. Your host for this session is Janani Ravi*. [Video description ends]

The functions that we've worked with so far have been useful, but not really that useful. We really need a way to customize our functions, and we do that using InputArguments.

[Video description begins] *A blank Jupyter notebook named InputArguments displays*. [Video description ends]

Well, what does that mean? Let's see, let's go back to our introduction function that we've seen before.

[Video description begins] *In code cell 1, she types the following lines of code: line 1: def my_introduction ( ): line 2: print ("Hello, my name is Harry") line 3: print ("I live and work in Dallas").* [Video description ends]

It says Hello, my name is Harry, I live and work in Dallas. Now, when we invoke this my_introduction function, this will give you Harry's introduction.

[Video description begins] *In code cell 2, she types the following code: my_introduction ( ). The following output displays below the code: line 1: Hello, my name is Harry line 2: I live and work in Dallas.* [Video description ends]

It's not really possible for us to use this function to introduce, say Lily, or James, or somebody else. Now, assume that all of the introductions are my name is something, and I live and work in some city. There should be a way to reuse the same function to specify different names and different cities, and that's exactly where input arguments are useful.

[Video description begins] *In code cell 3, she types the following lines of code: line 1: def introduction (name): line 2: print ("Hello, my name is", name) line 3: print ("I live and work in Dallas").* [Video description ends]

I'm going to define a new function here called introduction. Now, instead of just the parenthesis followed by the colon, observe that I have a variable within parenthesis, and that variable is called name.

This is an input argument to the introduction function, and this is where you can customize and parameterize your function. Specifying this input argument as a part of your function definition allows you to pass in your own custom name for every function invocation. Observe the first print statement within this introduction function.

It says Hello, my name is, and it references this name input argument that we've passed in. Input arguments passed into a function can be referenced within a function. Now that we've defined this function, let's go ahead and invoke this function. But this time, when you invoke this function you have to specify a value for this name input argument.

[Video description begins] *In code cell 4, she types the following code: introduction ("Lily").* [Video description ends]

I've invoked the introduction function and I've passed in Lily as the value for name. This is the equivalent of an assignment statement that I set name equal to Lily. And when the introduction function is executed, it'll printout Hello, my name is Lily, I live and work in Dallas.

We've managed to customize the same introduction, but this time the introduction is for Lily and not for Harry. Now, if you want somebody else to introduce himself or herself, let's say it's Henry, you simply invoke the introduction function and pass in a different value for the name input argument.

[Video description begins] *In code cell 5, she types the following code: introduction ("Henry").* [Video description ends]

Here, name is equal to Henry, and this invocation of the introduction function will print out Hello, my name is Henry, I live and work in Dallas. Now, we can go one step further. This introduction function only works for everyone who lives in Dallas. So we can have different names, but the city is the same. Now, let's define a function which allows individuals from different cities to introduce themselves.

[Video description begins] *In code cell 6, she types the following lines of code: line 1: def general_introduction (name, city): line 2: print ("Hello, my name is", name) line 3: print ("I live and work", city).* [Video description ends]

We want to be able to customize a name as well as city, and this is possible by specifying two input arguments when you define your function. Here is a function that I have defined called general introduction, it takes in two input arguments, name as well as city.

And within this function I simply print out Hello, my name is whatever is in the name variable, I live in work in whatever is in the city variable. Now, this is a function that can be used by any individual to introduce himself or herself.

[Video description begins] *In code cell 7, she types the following code: general_introduction ("Lily", "San Francisco").* [Video description ends]

Here is Lily from San Francisco, observe that the first input argument is the name that corresponds to the name input argument in our function definition. The second argument that we've specified corresponds to the city.

This is assigned to the city input argument in our function definition. Keep in mind that the order in which you specify the input argument is meaningful here. Lily will be assigned to name, San Francisco to city, which is why you get Hello, my name is Lily, I live and work in San Francisco. Let's use this general introduction function to introduce John.

[Video description begins] *In code cell 8, she types the following code: general_introduction ("John", "Dublin").* [Video description ends]

John lives and works in Dublin. So what you get printed out to screen Hello, my name is John, I live and work in Dublin. The name is the first input argument, city is the second. Now, if you remember, I said that the position of the values that you specify for your input arguments is significant.

[Video description begins] *In code cell 9, she types the following code: general_introduction ("Dublin", "John").* [Video description ends]

Now, here is the general introduction function that I have invoked. But the first value that I've specified is the name of the city and the second value is the actual name. Remember that functions don't know the semantics of what you're trying to convey.

What will happen here is that the value Dublin will be blindly assigned to the name input argument, and John to city. So what you'll get printed out to screen with this function is Hello, my name is Dublin, I live and work in John, which is kind of weird.

When you define input arguments to customize your function, make sure that you specify the values for these input arguments in the right order. That's because, by default, these input arguments are positional arguments.

And there are other kinds of arguments in Python as well, we'll come to that in a little bit. But remember, the order is important by default. Let's invoke a function that we wrote early on. This is a function which allowed no input arguments. We hadn't specified input arguments when we defined this function.

[Video description begins] *In code cell 10, she types the following code: my_introduction ("Lily"). A screenshot of the code in cell 1 displays on the right.* [Video description ends]

So I'm going to invoke my introduction by passing in a single input argument, Lily, and you'll find that this is an error in Python. This is a TypeError, my_introduction takes 0 positional arguments, but 1 was given.

Python tells you here that the function does not accept any positional arguments, but you still try to specify one, you cannot invoke the function this way. Another function that we had defined earlier was the introduction function. And if you remember, this took in just one input argument, that is name.

[Video description begins] *In code cell 11, she types the following code: introduction ("Lily", "San Francisco"). A screenshot of the code in cell 3 displays on the right.* [Video description ends]

Our function definition accepted just name, but we are trying to invoke it by specifying two input arguments, Lily and San Francisco, and this is an error as well.

You can see that the error is very clear indeed here, introduction takes just 1 positional argument based on its definition, but we try to invoke it using 2. Python does not accept this. Now, let's go on to our general introduction function. When we have defined this function, we had specified two input arguments, name as well as city.

[Video description begins] *In code cell 12, she types the following code: general_introduction ("Lily"). A screenshot of the code in cell 6 displays on the right*. [Video description ends]

But here you see that I'm trying to invoke it by specifying the value for just one input argument, Lily, which will automatically be assigned to the first variable, name. This is an error as well, general_introduction missing one required positional argument, city.

You can see that this message is fairly clear as well. Python tells you that there is a required positional argument named city for which you haven't specified a value. The value is missing, so this invocation is wrong.

If you've defined your custom function with a certain number of input arguments, make sure you specify the right number of input arguments when you invoke that function.

Input Arguments - Referencing Global Variables

[Video description begins] *Topic title: Input Arguments- Referencing Global Variables. Your host for this session is Janani Ravi*. [Video description ends]

Now that you know how to use input arguments to customize what a function works on, you can do some rather cool things. Here is a function called square that calculates the square of a number.

[Video description begins] *The InputArguments notebook displays. Its first 12 code cells are populated by code. In cell 13, she types the following lines of code: line 1: def square (x): line 2: print ("The square of", x, "is", x *x)*. [Video description ends]

The input argument that we pass in here is x, which is the number that we want to square. This function will simply print out the number itself, as well as the square of that number to screen. Now that we've defined this function, let's invoke this function and pass in the number 3.

[Video description begins] *In code cell 14, she types the following code: square (3)*. [Video description ends]

And this will helpfully tell us that the square of 3 is 9, which we know is right. Let's invoke this function once again with a huge number. This is 10,001. Well, for Python, it makes no difference and we get square of 10,001, it's this very large number that you see here on screen.

[Video description begins] *In code cell 15, she types the following code: square (10001)*. [Video description ends]

Now, our square function here assumes that whatever value you pass in for the variable x can be multiplied with itself. So if you try to calculate the square of James, well, that is an error. You can't multiply sequence by non-int type.

[Video description begins] *In code cell 16, she types the following code: square ("James")*. [Video description ends]

Now ideally, we would have wanted our function to check whether the input argument was of type int or float and only then perform the multiplication operation. That is error checking, and we'll discuss that later. So far,

we've been invoking our functions by specifying values for input arguments.

[Video description begins] *In code cell 17, she types the following lines of code: line 1: num= 25 line 2: square (num).* [Video description ends]

However, you can invoke functions by specifying variables that hold values. Here, I've assigned a variable, num. Its value is 25 here, and I'm going to invoke the square function and pass in the num variable.

And you'll see that when you invoke this function in this manner, Python will do the correct intuitive thing. It'll see what value is held within the num variable, and that is the value that is passed into the input argument of this function. So you'll get the square of 25 is 625. Observe how we're slowly getting our custom functions to be more powerful.

[Video description begins] *In code cell 18, she types the following lines of code: line 1: another_num= 100 line 2: square (num) line 3: square (another_num).* [Video description ends]

Here is another variable that we've initialized, another_num = 100, and I've made two calls or invocations to the square function. The first one passes in the num variable, and the second, another_num.

Now you can see that this function is called twice, once with the value 25, the square of 25 is 625. And the second time with the value of 100, which is stored in a another_num variable. Let's now understand a few nuances about global variables, that is, variables that are declared outside of a function and the input arguments of a function.

[Video description begins] *In code cell 19, she types the following lines of code: line 1: salary= 1000 line 2: expense= 300 line 3: def my_savings (a, b): line 4: print ("My total savings:", salary- expense).* [Video description ends]

Here I have a function defined called my_savings, and it takes in two input arguments, a and b. I've also assigned values to two global variables here, salary and expense. Salary is equal to 1,000 and expense is 300. Let's assume both of these are in dollars.

Now, within my my_savings function, I've printed out My total savings is salary - expense. Salary as well as expense here refers to the global variable salary and expense defined outside of the scope of the function.

Even though we have two input arguments, a and b, passed into my_savings, we aren't really using those input arguments, and you'll see the implications of that in just a bit. Let's go ahead and invoke my_savings with the value 1,000 and 300.

[Video description begins] *In code cell 20, she types the following code: my_savings (1000, 300).* [Video description ends]

The value of 1,000 will be assigned to the input argument a, so the variable a will contain 1,000 and the variable b will contain 300. When you invoke this function, you will get the result My total savings: 700. And you'll think that your function is perfectly defined and you're all set here.

But notice a detail, within our function, we're accessing the variables salary and expense, and not a and b. Now it so happens that the salary variable has been assigned to 1,000, which is the first input argument that we pass into my_savings.

And expense is equal to 300, which is the same as the second input argument that we pass into my_savings. So my_savings seems to work, even though it's not using the value of a and b, and instead using salary and expense directly.

Now, if you want to invoke the my_savings function and pass in $1,200 and $400 as the two input arguments, the result that you might expect is that your total savings is 1200 minus 400 equal to 800. But you still get my

total savings equal to 700.

[Video description begins] *In code cell 21, she types the following code: my_savings (1200, 400).* [Video description ends]

Something is clearly wrong here. What's wrong is in how you've defined your my_savings function. You want my_savings to operate on a and b, and not on some external global variables that happen to exist. So you need to edit this function to be like what you see here on screen, my_actual_savings.

[Video description begins] *In code cell 22, she types the following lines of code: line 1: def my_actual_savings (a, b): line 2: print ("My actual savings:", a-b).* [Video description ends]

my_actual_savings takes in two input arguments, a and b. It subtracts b from a, and that is your actual saving. Here, it assumes that a is your salary and b is your expense. The variables are not really named well here because you don't know what they represent.

However, we'll work with these for now. a - b is your actual savings. Now, when you invoke my_actual_savings with 1,200 and 400, you get the result that you expect, your savings are $800.

[Video description begins] *In code cell 23, she types the following code: my_actual_savings (1200, 400).* [Video description ends]

When you're writing functions in Python, this is something that you need to watch out for. If you expect to operate on the input argument, make sure that you reference the input arguments and not some other variables that happen to exist.

Now, your variable names for your input arguments should be meaningful, so that someone who's using your function knows what they represent.

[Video description begins] *In code cell 24, she types the following lines of code: line 1: salary= 2000 line 2: expense= 700 line 3: def calculate_savings (salary, expense): line 4: print ("My savings are:", salary-expense).* [Video description ends]

If you want to calculate savings and you're expecting the user to pass in salary and expense, name your input arguments salary and expense. Calling them a and b, x and y, John and Kathy, all of these are legit in Python, but really, it just makes your code more confusing.

You might have observed something interesting about the calculate_savings function that I have defined here. It takes in two input arguments. The names of these input arguments are salary and expense, and salary minus expense is our saving.

But observe that we have two global variables named salary and expense as well. And these have been assigned values 2,000 and 700. Now, you might be a little confused. Within calculate_savings, when you reference salary and expense, what variables are you referring to, the input arguments or the global variables?

I'll give you the answer first so that you're not confused and not in suspense. When you have input arguments which have the same name as external global variables, the input arguments are what are referenced within the function when you use that name. All right, let's demonstrate this. I'm going to invoke the calculate_savings functions without any input arguments at all.

[Video description begins] *In code cell 25, she types the following code: calculate_savings ( ).* [Video description ends]

Now, let's see if it references salary and expense, which are defined globally, and you can see that it doesn't.

It clearly says here in this error that two input arguments are required. It's pretty clear here that even though we have global variables with the same name, salary and expense, available for this function to use, the function doesn't actually reference those variables.

It expects the input arguments to be specified. All right, now I'm going to go ahead and invoke the calculate_savings function.

[Video description begins] *In code cell 26, she types the following lines of code: calculate_savings (salary, expense). A screenshot of the code in cell 25 displays on the right*. [Video description ends]

But this time, I'm going to specify values for the input arguments, and those values are the salary and expense global variables that we had set up earlier. Here, salary and expense refer to the global variables, where salary = 2,000, and expense = 700.

And this gives us My savings are: 1,300. Now, this might still be a little confusing. Where are we getting the values for salary and expense from, when we are inside the function, from the global variables, or from the input arguments? Now, this example will make things very clear.

[Video description begins] *In code cell 27, she types the following code: calculate_savings (3000, 1000)*. [Video description ends]

I'll invoke the calculate_savings function and pass in 3,000 and 1,000. 3,000 is the salary, 1,000 is the expense. And observe that the result here is 2,000. So even though the global variables were initialized to 2,000 and 700 for salary and expense, the values that we passed in in the input arguments of this function were what were used to calculate savings.

The value of 3,000 was assigned to the first input argument, salary, 1,000 was the expense, that is the second input argument. And the operation within the function, salary - expense, was performed on the values that we passed in as input arguments to calculate savings. Let's see another example that ought to make this clear.

[Video description begins] *In code cell 28, she types the following lines of code: line 1: total_salary= 4000 line 2: total_expense= 2500 line 3: def calculate_savings (total_salary, total_expense):*. [Video description ends]

Here, I've defined two new variables, total_salary is equal to 4,000 and total_expense is equal to 2,500. And I'm invoking the calculate_savings function by passing in the variables total_salary and total_expense. What will be the output of the function? Well, you'll see that the result is 1,500. calculate_savings will use the values that you have passed in as input arguments to the function, not the global variables.

Input Arguments - Using Positional Arguments

[Video description begins] *Topic title: Input Arguments- Using Positional Arguments. Your host for this session is Janani Ravi*. [Video description ends]

Now that we're a little comfortable with the idea of input arguments. We can use input arguments and write functions which do a variety of different things. Here is a function that takes in two input arguments, a string and the number of times that string should be printed out to screen.

[Video description begins] *The InputArguments notebook displays. In code cell 29, the host types the following lines of code: line 1: def print_many_times (string, times): line 2: for i in range (times): line 2: print (string)*. [Video description ends]

Now, within this function, all I'm doing is running a for loop in the range, which is specified by the times input argument. And I invoke the print statement as many times as specified. Now, let's invoke this print_many_times function with Skillsoft and 5. And you'll see the output here is that Skillsoft is printed out five times.

[Video description begins] *In code cell 30, she types the following code: print_many_times ("Skillsoft", 5).* [Video description ends]

Remember that the input arguments that you specify are positional in nature. If you invoke print many times by passing in 5 for your first input argument and Skillsoft for the second. You'll find that this results in an error because we'll try to set up a for loop that iterates Skillsoft number of times. That makes no sense.

[Video description begins] *In code cell 31, she types the following code: print_many_times (5, "Skillsoft").* [Video description ends]

Whether it's built in functions that you invoke or custom functions, make sure you pass in your input arguments in the right order. Now let's see another example of the print_many_times function. This time, we'll specified documentation for this function.

[Video description begins] *In code cell 32, the host types the following lines of code: line 1: def print_many_times_with_doc (string, times): line 2: for i in range (times): line 2: print (string). Between lines 1 and 2, many lines of documentation display.* [Video description ends]

The documentation is more detailed here, it will exactly specify to users of this function. What is expected in the string input argument as well as the times input argument.

You can see that there is first a general description of what this function does, and then we have a parameters portion in the documentation. Where we specify the string and times parameter and the kind of values that we expect in these input arguments.

There's also a statement here which says this function has no return value. We'll talk about return values in just a bit, you can just ignore it for now. Now that you define this print_many_times_with_doc function. You can access the documentation using the __doc__ attribute of your function object.

[Video description begins] *In code cell 33, the host types the following lines of code: def print_many_times_with_doc, __doc__.* [Video description ends]

When you view the documentation here as the result of your code execution. You can see that it contains \ns, which are not identified by Jupyter Notebook. You should actually print out your documentation using the print statement and it will be printed out nicely formatted as you can see here on screen.

[Video description begins] *In code cell 34, the host types the following lines of code: def print (print_many_times_with_doc, __doc__).* [Video description ends]

Let's get a little more practice with functions and input arguments here. Here is a function that takes in two numbers, a and b, and prints out which of these numbers have a higher value. It prints out the higher number.

You can see that there is an if conditional check within this function. If a is greater than b, we'll say that the higher number is a, else we'll say that the higher number is b.

[Video description begins] *In code cell 35, the host types the following lines of code: line 1: def print_higher_number (a, b): line 2: if a > b: line 3: print ("Higher number is", a) line 3: else: line 4: print ("Higher number is", b).* [Video description ends]

Now, we'll pass in any numbers of our choice to the print_higher_number function. I'm going to pass in 10 and 5, and I expect the result, the higher number is 10, and that's exactly what is displayed here.

[Video description begins] *In code cell 36, the host types the following code: def print_higher_number (10, 5).* [Video description ends]

Observe that with no thought at all, you've actually written a function that is useful. This is the kind of function that you might write when you're working with numbers in data science.

[Video description begins] *In code cell 37, the host types the following code: def print_higher_number (10, 20).* [Video description ends]

Print the higher number. The higher number here is clearly 20, between 10 and 20. Now, it's quite possible that while defining your function, you introduce some kind of error. Now remember, if you have multiple code paths within your function, such as within this if-else condition.

There is a code path under the if, and there's a code path for the else. Make sure that you test all code paths so that you debug your function thoroughly and remove all errors.

[Video description begins] *In code cell 38, the host types the following lines of code: line 1: def print_higher_number_with_error (a, b): line 2: if a > b: line 3: print ("Higher number is", a) line 3: else: line 4: print ("Higher number is", b) line 5: result= b + 'a'.* [Video description ends]

Observe that in the else block of this function, I've tried to add a string and a number, and this is clearly an error. Now if I were to invoke this function with 10, 5, I'm only going to hit the code that is in the if block.

[Video description begins] *In code cell 39, the host types the following lines of code: line 1: def print_higher_number_with_error (10, 5).* [Video description ends]

The code here has no error, the higher number is 10, 10 was passed in into the a input argument, a is greater than b. In our if block, there was no error and it seems like this function works fine.

But if you want to invoke this function with 10, 20, b is greater than a, which means if a greater than b is not satisfied, we move into the else block. The code that we've written in the else block of our function has an error. That's when this error is encountered.

[Video description begins] *In code cell 40, the host types the following lines of code: line 1: def print_higher_number_with_error (10, 20).* [Video description ends]

So make sure that when you write code which has multiple parts, test all of your parts. So far, we've defined functions to have one or two input arguments. There is no constraint on the number of input arguments that you can specify for a function.

[Video description begins] *In code cell 41, she types the following lines of code: line 1: def multiply (num_1, num_2, num_3): line 2: print ("Multiplication result: ", num_1 * num_2 * num_3).* [Video description ends]

Here is an example of the multiply function with three input arguments, num_1, num_2 and num_3. The code within the function performs a simple multiplication operation. And all three numbers are multiplied together and the result is printed out to screen.

Now, when you invoke this function, you'll have to specify values for all three input arguments. Otherwise, you'll see an error. Observe that the input arguments that I've specified here are integers as well as floats.

[Video description begins] *In code cell 42, she types the following code: multiply (1, 2.3, 4).* [Video description ends]

And that's fine because integers as well as floats can be multiplied together. Here is our multiplication result. I'll now invoke the multiply function with a different set of input arguments, but observe that I've specified one additional argument. I've specified four arguments here, our function only accepts three.

[Video description begins] *In code cell 43, she types the following code: multiply (1, 2.3, 4, 10).* [Video description ends]

And this is also an error. Multiply takes three positional arguments, but four were given. Functions can accept input arguments of any type of data. They can be primitive types such as floating points, strings, integers.

You can have complex types passed in as input arguments to functions. Functions can also accept custom data types, those that you defined using custom classes. Don't worry about that yet, you'll soon learn it. Here I have a function named length and it takes as its input argument just one, a list.

It then iterates over the list and counts the number of elements that are present in the list and prints out to screen. The length of the list is and whatever it has calculated.

[Video description begins] *In code cell 44, she types the following lines of code: line 1: def length (some_list): line 2: count= 0 line 3: for element in some_list: line 3: count +=1 line 4: print ("The length of the list is", count)*. [Video description ends]

Now this is similar but not exactly the same as the built-in len function. Once you've defined this length function, you can invoke it. And as an input argument, you can pass in any list.

Here I've defined a list of numbers called num_list. And this num_list is what I pass in as an input argument to the length function. And it tells me correctly that the length of this list is 7.

[Video description begins] *In code cell 45, she types the following lines of code: line 1: num_list= [4, 6, 12, 7, 2, 9, 23] line 2: length (num_list)*. [Video description ends]

You'll see that the length function that we've defined works with all kinds of lists. Here is a list of cars where the elements are strings, which are brand names of cars. I'll pass in this cars_list to the length function that we've defined. And it tells me that the length of this list is 4, which is once again correct.

[Video description begins] *In code cell 46, she types the following lines of code: line 1: cars_list= ["Toyota", "Honda", "Hyundai", "Ford"] line 2: length (cars_list)*. [Video description ends]

You can also define a list on the fly and pass that into the length function. Observe that here, length does not accept a variable. But instead we've defined the list directly within square brackets, and passed that list into the length function as an input argument. And we got the right result, the length of this list is 3.

[Video description begins] *In code cell 47, she types the following code: length ([4.5, 7.8, 3.3])*. [Video description ends]

Now you might feel, and you wouldn't be very wrong, that this length function that we just defined is very similar to the built-in len function. But there is a subtle but important difference.

When you use the built-in len function, you can invoke len on say the cars list and assign whatever result that you get to a variable. This is the num_cars variable, and you can see that the value that num_cars holds is 4. That is the number of elements in the cars_list.

[Video description begins] *In code cell 50, she types the following lines of code: line 1: num_cars=len (cars_list) line 2: num_cars. A screenshot of the code in cell 46 displays on the right*. [Video description ends]

What if we try to do the same operation using our length custom function?

[Video description begins] *In code cell 51, she types the following code: num_cars=length (cars_list)*. [Video description ends]

So I'm going to say num_cars = length(cars_list). And I'm going to check what value is present within num_cars. I'm going to print out the value of num_cars to screen and you'll see that there is nothing.

[Video description begins] *In code cell 52, she types the following code: num_cars*. [Video description ends]

Clearly, there's something missing in the length custom function that we've defined. num_cars does not hold any value, I'm going to call print on num_cars. And you can see that it prints out None rather than 4. What is that something missing? Well, that is a return value from our function and that's what we'll discuss next.

[Video description begins] *In code cell 53, she types the following code: print (num_cars)*. [Video description ends]

Return Values - Functions

[Video description begins] *Topic title: Return Values- Functions. Your host for this session is Janani Ravi*. [Video description ends]

You know what would make functions really useful? If they performed some calculation and return the result to us. And that's exactly what we'll study here in this demo. How do we get functions to return values.

[Video description begins] *A blank Jupyter notebook named ReturnValues displays*. [Video description ends]

Here is an example of a function from which you might expect a result. This function is called subtract, and it takes in two input arguments, num_1 and num_2. And it subtracts num_2 from num_1, and stores the result in a variable result.

[Video description begins] *In code cell 1, she types the following code: line 1: def subtract (num_1, num_2): line 2: result= num_1 - num_2*. [Video description ends]

Now that we've defined this function, let's invoke it specifying some input arguments, subtract(10, 7.7).

[Video description begins] *In code cell 2, she types the following code: subtract (10, 7.7)*. [Video description ends]

What we would expect as a result from the subtraction operation is 2.3, 10 minus 7.7. But you see that this subtraction operation displays nothing to screen. What if you try to store the result of a subtraction in a variable, r? r = subtract(100, 50).

[Video description begins] *In code cell 3, she types the following code: r= subtract (100, 50)*. [Video description ends]

When we work with built-in functions in this manner, we'll expect that the result of the subtraction would be available in the variable r. Let's perform the subtraction and let's print r out to screen and you'll see that r has None instead of the value 50. 100 minus 50 we know is 50 and I really expected that r would have the value 50, but it has the value None.

[Video description begins] *In code cell 4, she types the following code: print (r)*. [Video description ends]

If you've never encountered this None value in Python, you might be a little surprised. There is nothing to it. None is a special value in Python which indicates no value present or just nothing. Basically, r function did not return the result of the subtraction, which is why r has None.

The None special value has its own special type. So if you print out the type of this r variable, you'll see that it is of type NoneType.

[Video description begins] *In code cell 5, she types the following code: type (r)*. [Video description ends]

This is the special type used to represent the special value None. Why is it that the subtract function returned this special value? Well, it's because we hadn't specified that subtract returns anything.

And by default, all functions return none when you haven't explicitly specified a return value. Let's see what I mean here. I'm going to define this new custom function here called subtract_returns_none_by_default.

And it takes in two input arguments, num_1 and num_2. And it performs the subtraction exactly as before. But observe that I have this keyword here, return, at the very end of this custom function. And I return None.

[Video description begins] *In code cell 6, she types the following code: line 1: def subtract_returns_none_by_default (num_1, num_2): line 2: result= num_1 - num_2. line 3: return none*. [Video description ends]

Now, return is a special keyword that Python understands, just like the def keyword to define a function. The return keyword allows you to return a value from the function and this value can be any kind of information.

I've specified return None here because this is the default return value when you do not specify a return statement within your function. Now that we have this function defined, let's go ahead and invoke it, subtract_returns_none_by_default.

Pass in 100 and 50, and store the return value in the variable r. Now when you print out r to screen, you'll see that r is None.

[Video description begins] *In code cell 7, she types the following code: line 1: r= subtract_returns_none_by_default (100, 50) line 2: print (r). A screenshot of the code in cells 3 and 4 displays on the right*. [Video description ends]

This is the exact same result that we got earlier with the plain subtract function.

[Video description begins] *In code cell 8, she types the following code: type (r)*. [Video description ends]

If you see the type of r, you'll see that it's the special None type. Now what we actually want is for this subtract operation to return the subtraction result. And here is how we can modify our function to do this.

[Video description begins] *In code cell 9, she types the following code: line 1: def subtract_returns_a_result (num_1, num_2): line 2: result= num_1 - num_2. line 3: return result*. [Video description ends]

Here is a new function that I've defined called subtract_returns_a_result. It takes in num_1 and num_2, calculates the result num_1 - num_2. And then I use the return keyword to return this result to whoever has invoked this function. Let's now perform the subtraction using this function with the return value that we just defined.

[Video description begins] *In code cell 10, she types the following code: line 1: r= subtract_returns_a_result (100, 50) line 2: print (r)*. [Video description ends]

r = subtract_returns_a_result, (100, 50) are our input, arguments. I'm going to print out r to screen, and you get what you finally expect, the value of 50, the result of our subtraction. Input arguments are what you use to pass in information to a function.

Return values are what you use to get information from a function. We'll now go back to a function that we've written before. This is the length function which takes as an input argument a list, and then counts the number of elements that are present in the list. This is the same length function as before, except at this time we return count from this length.function.

[Video description begins] *In code cell 11, she types the following lines of code: line 1: def length (some_list): line 2: count= 0 line 3: for element in some_list: line 4: count +=1 line 5: print ("The length of the list is", count) line 6: return count. A screenshot of the code for the length function from the previous demonstration displays on the right*. [Video description ends]

Count holds the number of elements in the list. This length function now has a return value. And now you'll see that we'll be able to write statements such as these. l = length, and then you pass in a list as the input argument.

[Video description begins] *In code cell 12, she types the following lines of code: l= length ([100, 200,300]):* [Video description ends]

When this bit of code is executed the print statement within your length function is also executed, you see that the length of this list is 3. And if you print out the value of l here, you'll see that l contains the value 3.

[Video description begins] *In code cell 13, she types l.* [Video description ends]

The return count from the length function, the number of elements in this list is 3, which is now stored in the variable l.

[Video description begins] *In code cell 14, she types the following code: print (l).* [Video description ends]

The return value from our function is now available in this variable. Now that we know how to pass information to a function and get information from a function, you can write more complex functions such as this.

[Video description begins] *In code cell 15, she types the following code: line 1: def find_max_in_list (some_list): line 2: max_element= some_list [0] line 3: length= len (some_list) line 4: for i in range (i, length): line 5: if some_list (i) > max _element: line 6: max_element = some_list (i) line 7: return max_element.* [Video description ends]

Here is a function that I've defined to find the maximum element in the list. I've given the function a meaningful name, find_max_in_list. The input argument that I pass in is a list of any kind of elements as long as the elements are comparable using the greater than operation.

Now I have no idea what the maximum element in the list is. So I'm going to initialize the max_element variable to the first element in the list. I'll then calculate the length of the list by using the built-in len function.

I've set max_element to be equal to the first element in the list. Then I'll run a for loop through the entire length of the list and check every element starting from the second element in the list.

If the element is greater than the current max_element that I have stored in the max_element variable, then I update the value stored in the max_element variable. If some_list [i] > max_element, max_element = some_list [i].

Once this for loop is complete, whatever value is stored in the max_element variable is the maximum element in the list that we passed in to this function. And this is the value that I'll return to the caller of this function.

The caller of this function is basically the code that invoked this function. Here is a function that goes beyond the very simple. So I suggest you pause and kind of understand what exactly it does before you move on. Once we've defined this function, let's set up a list with the elements in jumbled order.

[Video description begins] *In code cell 16, she types the following code: num_list= [5, 4 , 7, 2, 10, 11, 3].* [Video description ends]

This is the num_list, which is just a list of integers. And I'm going to use this find_max_in_list custom function that I just defined to find the maximum element in this list.

[Video description begins] *In code cell 17, she types the following code: line 1: max_element= find_max_in_list (num_list) line 2: print (max_element).* [Video description ends]

I invoke or call the find_max_in_list function, pass in the num_list, and store the return value in max_element. And when I print out max_element to screen, I get the result 11, which is correct. That is the largest number in

this list. Let me modify num_list a little bit. I'm going to append the value 24 to num_list.

[Video description begins] *In code cell 18, she types the following code: num_list.append (24)*. [Video description ends]

And I'm going to invoke find_max_in_list once again and pass in this new num_list which has 24 appended onto the very end. And when I print out the value of max_element, you can see here that we get the correct result. 24 is now the largest element in num_list.

[Video description begins] *In code cell 19, she types the same code as in cell 17*. [Video description ends]

We've seen that the return keyword is what we use to return information from a function.

[Video description begins] *In code cell 20, she types the following code: line 1: def empty_return (x, y): line 2: total = x + y line 3: return*. [Video description ends]

In Python it's possible to use the return keyword to return from a function without specifying a corresponding value, as you see here on screen.

This is a function that has an empty_return. The empty_return function takes in two input arguments, x and y, calculates the sum of these two, and then simply returns. It does not include a return value. A return statement with no return value is like a returning None.

[Video description begins] *In code cell 21, she types the following code:result= empty_return (10, 20)*. [Video description ends]

So if I invoke the empty_return function and store the return result in the result variable, and I printout result to screen, you'll see that it contains the value None.

[Video description begins] *In code cell 22, she types the following code: print (result)*. [Video description ends]

We've already discussed earlier that None is a special value that Python understands and None belongs to the special data type NoneType.

[Video description begins] *In code cell 23, she types the following code: type (result)*. [Video description ends]

Return Values - Multi-return Statements

[Video description begins] *Topic title: Return Values- Multi-return Statements. Your host for this session is Janani Ravi*. [Video description ends]

It's not just primitive simple types that you can use as return values from functions. You can return more complex information as well. In fact, you can return more than one bit of information as you see here on screen.

[Video description begins] *The ReturnValues notebook displays. In code cell 24, the host types the following code: line 1: def add_sub (x, y): line 2 : add_result= x + y line 3: sub_result= x - y line 4: return add_result, sub_result*. [Video description ends]

I've defined a new function here called add_sub, which takes in two input arguments, x as well as y. Within the function body, we perform two separate operations and calculate two different results. We have the add_result, which is the result of adding x and y.

And we have the sub_result, which is the result of subtracting y from x. We can return both of these results in one go using a single return statement.

We return add_result, sub_result. When we have two return values from a function which are separated using a comma, what does that mean? So I'm going to invoke add_sub and passing the input arguments 5, 4 and you can see that what it returns is a tuple.

[Video description begins] *In code cell 25, she types the following code: add_sub (5, 4).* [Video description ends]

The first field in the tuple is 9. 9 is the result of adding 5 and 4. 5 plus 4 is equal to 9. 1 is the result of subtracting 4 from 5. 5 minus 4 is equal to 1. When we have multiple return values, they're collected together and returned in the tuple format.

[Video description begins] *The output is (9, 1).* [Video description ends]

When you return multiple values separated using commas and this can be any number, not just 2, 3, 4, 5, 6, 10. All of these values will be wrapped up into a tuple and returned as a tuple. The tuple that is returned from this function can be stored in a variable as well. Here, result = add_sub(5, 4). And if you view result, you'll see that it is the tuple 9, 1.

[Video description begins] *In code cell 26, she types the following code: result= add_sub (5, 4) line 2: result.* [Video description ends]

If you want the individual values to be assigned to different variables, Python offers this very interesting syntax. Observe that we have result_1, result_2 on the left hand side of the assignment statement.

[Video description begins] *In code cell 27, she types the following code: result_1, result_2= add_sub (5, 4).* [Video description ends]

We know that the tuple returned from the add_sub function has two fields, the addition result and the subtraction result. The addition result that is the first field will be stored in the result_1 variable.

And the second field that is the subtraction result will be stored in the second variable. When you have comma separated variables on the left hand side of your assignment. Python will automatically unpack the tuple that was returned and assign the individual values to individual variables.

[Video description begins] *In code cell 28, she types the following code: result_1.* [Video description ends]

You'll see when you print out the value stored in result_1 that it is equal to 9, and the value stored in result_2 is 1.

[Video description begins] *In code cell 29, she types the following code: result_2.* [Video description ends]

So if you have a function with multiple return values. The individual return values can be assigned to individual variables directly as in this assignment statement. Now, let's say you're invoking the add_sub function, but you're only interested in the addition result. You're not interested in the subtraction result.

[Video description begins] *In code cell 30, she types the following code: result_1, _ = add_sub (5, 4) line 2: result_1.* [Video description ends]

Observe the left hand side of this assignment statement, result_1, _. This means that you're ignoring the second field of the tuple that is returned which is the subtraction result in our case. What you're indicating to Python with this statement is you're interested in the first result.

You'll store that in the result_1 variable. You want to ignore the second field that is returned, so you use the _ to indicate that. result_1 will contain 9, you can't access result_2. The underscore basically is a way of ignoring a return value.

Let's say you want to ignore the first field in the tuple that is returned, you'll specify _, result_2 in the left-hand side of your assignment statement. The second return value from the add_sub function that is the result of the subtraction will be stored in result_2. We've ignored the result of the addition as indicated by the underscore on the left-hand side our assignment.

[Video description begins] *In code cell 31, she types the following code: _, result_1 = add_sub (5, 4) line 2: result_2*. [Video description ends]

Just like you can have multiple written values from a function, you can have multiple return statements from a function as well. And each return statement can return a different value.

[Video description begins] *In code cell 32, she types the following code: line 1: def positive_or_negative (sum): line 2: if sum> 0: line 3: return "Positive!" line 4: else: line 5: return "Zero or negative"*. [Video description ends]

Here is a function that we've defined, a custom function which checks to see whether a number is positive or negative. It takes in exactly one input variable, num. And if num is greater than 0, we return positive, that's the string.

Otherwise, we return 0 or negative. Now the return from this function will be different depending on what code path is executed. The code within the if block or the code within the else block. Now if we invoke this function and pass in the number 20, we know that's a positive value. So if num greater than 0 will evaluate to true, and this function will return positive.

[Video description begins] *In code cell 33, she types the following code: positive_or_negative (20)*. [Video description ends]

Now if we invoke this function with -10, if num greater than 0 will evaluate to false. The else block will be executed and this function returns 0 or negative.

[Video description begins] *In code cell 34, she types the following code: positive_or_negative (-10)*. [Video description ends]

There is no limit on the number of return statements that you can specify within a function. Here is a function which checks for positive numbers, negative numbers, and also checks to see whether a number is 0.

[Video description begins] *In code cell 35, she types the following code: line 1: def positive_negative_zero(num): line 2: if num> 0: line 3: return "Positive!" line 4: elif num < 0: line 5: return "Negative!" line 6: else: line 7: return "Zero"*. [Video description ends]

There are three specific conditions that we are checking here. This corresponds to three return statements within this function. If number is greater than 0, we return positive. Otherwise, if number is less than 0, we return negative, else the number is actually equal to 0, so we return 0.

If you invoke this function using a positive number, let's say 90 as you see here in this example. num greater than 0 will evaluate to true and the function will return positive.

[Video description begins] *In code cell 36, she types the following code: positive_negative_zero(90)*. [Video description ends]

If you invoke this function with a negative number, let's say -90, the elif condition num less than 0 will evaluate to true. And the return value from this function will be the string negative.

[Video description begins] *In code cell 37, she types the following code: positive_negative_zero(-90)*. [Video description ends]

Now if you invoke this function with 0, the if and elif conditions will evaluate to false. And the code within the last else block is what will be executed, and the return value of this function will be 0.

[Video description begins] *In code cell 40, she types the following code: positive_negative_zero(0).* [Video description ends]

Now if you have a complex function with multiple return statements. It's often very easy to forget a return based on a condition that you're evaluating. Observe this function that I've defined here, positive_negative_zero_forgotreturn.

[Video description begins] *In code cell 41, she types the following code: line 1: def positive_negative_zero_forgotreturn(num): line 2: if num> 0: line 3: return "Positive!" line 4: elif num < 0: line 5: return "Negative!"* [Video description ends]

Here, I've checked for two conditions, num greater than 0 and num less than 0. But I've forgotten to check to see whether num is equal to 0, which means there is one case where this function will not return anything.

And this can be dangerous when you're working with production level code. When you test this function out, you'll find that it works perfectly fine for positive numbers, it'll turn positive.

[Video description begins] *In code cell 42, she types the following code: positive_negative_zero_forgotreturn(-10).* [Video description ends]

It works perfectly fine for negative numbers, the return will be negative. But when you test it out with 0, the function will return nothing. It'll return none.

[Video description begins] *In code cell 43, she types the following code: positive_negative_zero_forgotreturn(0).* [Video description ends]

Now this might be what you want from the function but it's often the case that you've simply forgotten a return statement. So when you use functions with multiple return statements, make sure that you test every bit of code that might be executed within your function.

Return Values - Complex Data Types

[Video description begins] *Topic title: Return Values- Complex Data Types. Your host for this session is Janani Ravi.* [Video description ends]

We're working in the same notebook, ReturnValues. So I'm going to go ahead and invoke a function that we had set up earlier that we had defined early on in this node. The find_max_in_list function, and as an input argument, I'm going to pass in an empty_list.

[Video description begins] *The ReturnValues notebook opens. In code cell 43, the host types the following code: line 1: empty_list = [ ] line 2: find_max_in_list (empty_list). A screenshot of the code for find_max_in_list definition displays on the right.* [Video description ends]

If you look at the code for this find_max_in_list function, you'll find that it assumes that the input list has at least one element. This is what we use to initialize the max_element variable.

Now when we pass in an empty list, it tries to access the first element in an empty list, which is clearly an error, list index out of range. Now, instead of having your function throw this error, it would be better if your function handle this in a more graceful manner. And that's exactly what we'll do. I'm going to redefine the function find_max_in_list.

[Video description begins] *In code cell 44, she types the following code: line 1: def find_max_in_list (some_list) : line 2: if len (some_list) == 0: line 3: print ("Zero element list!") line 4: return None line 5: max_element = some_list [0] line 6: length = len (some_list) line 7: for i in range (1, length): line 8: if some_list [i] > max_element: line 9: max_element = some_list [i] line 10: return max_element*. [Video description ends]

Now most of the code for this function remains exactly the same as before. I've made a few changes at the very beginning. I'm doing a conditional check to see whether the length of the list that we passed into this function is equal to zero.

If it is, I'm going to print out a message saying zero element list and return none from this function. The code in the if block will be executed only if the list that we pass in is an empty list. In the case of non-empty list, we'll go ahead and find the highest or the maximum element in the list as we did before. Now let's invoke this function find_max_in_list and pass in the empty list as we did before.

[Video description begins] *In code cell 45, the host types the following code: find_max_in_list (empty_list)*. [Video description ends]

Now instead of throwing an error, which was the result that we got previously, we get this nice message printed out to screen saying zero element list. You might find that this is a much more graceful way to handle errors.

Explicitly check for those errors and return values that makes sense. As I've mentioned before, the functions that you can define can contain arbitrarily complex code. Here is a function that performs something a little interesting. It tries to find the first capital letter in the string that you pass in as an input argument.

[Video description begins] *In code cell 46, she types the following code: line 1: def find_first_capital_letter (some_string): line 2: capital_letter= None line 3: for ch in some_string: line 4: if ch.upper ( ) == ch and ch != " ": line 5: capital_letter = ch line 6: break line 7: if capital_letter is None: line 8: return "No capital letters found" line 9: else: line 10: return "First capital letter" + capital_letter.* [Video description ends]

Within this function I initialize a variable called capital_letter to be none, indicating that we've not found an uppercase or capital letter so far. I'm then going to set up a for loop which iterates over every character in the string that we've passed in at the input.

For every character in some_string, I store the character in the ch variable. And I convert that character to uppercase and see whether the character in uppercase is equal to the character itself. That means this character is a capital letter.

I'm going to check for one additional condition here in my if statement. I'm checking to see whether the character is a blank character, because the upper case of a blank character is the blank character itself. I want to ignore blank characters in my check.

I'll iterate over the input string sequentially. Once I find a capital letter, I'll assign that character to the capital letter variable and break out of the for loop. Once we're out of the for loop, I'll check to see whether capital letter is none.

Is it equal to the none special value? If that's the case, then no capital letters were found in our input string. If the capital letter variable contains something other than none, this means that we have discovered an uppercase or capital letter in our input string.

And we return the first capital letter is whatever capital letter we've found. Let's take this function for a test drive. Find first capital letter, and the string that we pass in is how Are you, the letter A in are is what is capitalized.

[Video description begins] *In code cell 47, she types the following code:find_first_capital_letter (" how Are you")*. [Video description ends]

And our function rightly returns that the first capital letter is the letter A. Let's invoke this function once again, with all letters in lowercase, and you can see that our function says no capital letters found.

[Video description begins] *In code cell 48, she types the following code:find_first_capital_letter (" how are you").* [Video description ends]

Just like input arguments to a function can be of any data type, return values from a function can also be of any data type. You can have a function return a dictionary, for example.

[Video description begins] *In code cell 49, she types the following code: line 1: def create_dictionary_representation (name, age, occupation): line 2: dictionary = { line 3: "name" : name, line 4: "age" : age, line 5: "occupation" : occupation line 6: } line 7: return dictionary.* [Video description ends]

Here is a function called create_dictionary_representation. It takes in three input arguments, name, age and occupation, and creates a dictionary holding the name, age and occupation, along with its corresponding keys. And we return this dictionary from this function.

Now we can invoke create_dictionary_representation and store the result in a variable. This is the info_dictionary variable and you can see that the info_dictionary variable contains the input arguments in dictionary format.

[Video description begins] *In code cell 50, she types the following code: info_dictionary = create_dictionary_representation ("John" , 35, "Engineer"). In code cell 51, she types info_dictionary.* [Video description ends]

I'll now show you another example of a function that returns a complex data type. This time we'll return a list. This function is called generate_list, and it takes in two input arguments, name and number of elements.

[Video description begins] *In code cell 52, she types the following code: line 1: def generate_list (name, num_elements): line 2: return_list= [ ] line 3: for i in range (num_elements): line 4: return_list.append (name) line 5: return return_list.* [Video description ends]

This generate_list function returns a list, where the name that you pass in is repeated num_elements number of times. And the return value is this return list.

So we initialize the return list to an empty list initially, then set up a for loop to iterate num_elements number of times and each time we append name to this return_list. And finally, this return_list is what is returned to the caller or invoker of this function. With this function defined, let's call generate_list with the name Iris, and we want Iris to be repeated five times.

[Video description begins] *In code cell 53, she types the following code: line 1: some_list = generate_list ("Iris", 5) line 2: some_list.* [Video description ends]

We store the list that is returned from the function in the some_list variable. And if you look at some_list, you'll see that it contains the name Iris five times. Let's us generate_list on the name Louise, and we want this name to be repeated three times. And that's exactly what we get here in the result.

[Video description begins] *In code cell 54, she types the following code: line 1: some_list = generate_list ("Louise", 3) line 2: some_list.* [Video description ends]

What we just saw earlier using the for loop was the long-winded way of creating a list. You can also create a list using list comprehension. Here is our return_list where the name is repeated num_elements number of times.

But this time we use list comprehension in order to generate this list. Within square brackets we specify name for i in range(num_elements), which means name will be repeated num_elements number of times.

[Video description begins] *In code cell 55, she types the following code: line 1: def generate_list (name, num_elements): line 2: print ("Generate list using comprehension") line 3: return_list= [name for i in range (num_elements)] line 4: return return_list*. [Video description ends]

And this will be a list and this is what we return to the invoker or caller of this function. Let's invoke this generate_list function, and pass in Daisy and 4, and you can see that the returned list has Daisy four times.

[Video description begins] *In code cell 56, she types the following code: line 1: some_list = generate_list ("Daisy", 4) line 2: some_list*. [Video description ends]

You can make this generate_list function even more compact. You can specify your calculation within the return statement itself. Observe that I've moved the list comprehension statement, right into the return statement.

[Video description begins] *A screenshot of the code in cell 55 displays on the right. In code cell 57, she types the following code: line 1: def generate_list (name, num_elements): line 2: return [name for i in range(num_elements)]*. [Video description ends]

I haven't used a separate variable to store the result of the list comprehension. I've simply returned the list to the caller of this function and this works as well. Let's call generate_list, I want sunflower to be repeated four times and you can see that some_list has the element sunflower repeated four times. This just shows you that you can include calculations within your return statement as well.

[Video description begins] *In code cell 58, she types the following code: line 1: some_list = generate_list ("Sunflower", 4) line 2: some_list*. [Video description ends]

You can have your custom functions invoke other custom functions that you've defined earlier. I'm going to define four functions at a go to add, subtract and multiply two numbers. And these two numbers are passed in as input arguments a and b.

[Video description begins] *In code cell 59, she types the following code: line 1: def add (a, b) : line 2: return a + b line 3: def sub (a, b) : line 4: return a - b line 5: def mul (a, b) : line 6: return a * b line 7: def div (a, b) : line 8: return a / b*. [Video description ends]

You can see that I've kept each of these functions very compact and concise. The return statements also perform the calculation a plus b, a minus b, a multiplied by b and a divided by b. I'm now going to define an additional function here called calculate. This takes in three input arguments, a, b, and an operator.

And the operation that this calculate function will perform will depend on what you pass in in the operator argument. If the operator that you pass in is equal to the add string, then we'll add a and b and observe how we invoke the add custom function from within the calculate custom function.

So if operator is equal to the add string, we'll add a and b. If operator is equal to the sub string, we'll subtract b from a. We invoke the sub custom function. If operator is equal to mul, we'll multiply the two numbers. If operator is equal to div we'll divide a by b.

[Video description begins] *In code cell 60, she types the following code: line 1: def calculate (a, b, operator) : line 2: if operator == "add" : line 3: return add (a, b) line 4: if operator == "sub" : line 5: return sub (a, b) line 6: if operator == "mul" : line 7: return mul (a, b) line 8: if operator == "div" : line 3: return div (a, b)*. [Video description ends]

Observe how in each case the calculate function invokes add, sub, mul or div within the return statement. This means that the function will be invoked, and whatever value the function returned, that is what is returned as the return value for the calculate function. Let's go ahead and invoke the calculate function here. A is equal to 10, b is equal to five and the operation that I want to perform is the addition operation, and that's what I get here in my result. A + b = 10 + 5 which is 15. Let's invoke the calculate function once again.

[Video description begins] *In code cell 61, she types the following code: calculate (10, 5, "add")*. [Video description ends]

This time I want to perform the multiplication operation on 10 and 5, that gives me the result 50. Observe how we've cleverly customized the same function to perform different arithmetic operation.

[Video description begins] *In code cell 62, she types the following code: calculate (10, 5, "mul")*. [Video description ends]

Now we can perform division as well, 10 divided by 5 is 2.0. And you can also see how you perform the subtraction operation if you pass

[Video description begins] *In code cell 63, she types the following code: calculate (10, 5, "div")*. [Video description ends]

in sub. But what if you pass in an operator that our custom function does not understand, such as the meaningless operator?

[Video description begins] *In code cell 64, she types the following code: calculate (10, 5, "meaningless")*. [Video description ends]

Well, you'll get no result. Our calculate function did not understand what this operator was about. None of the if conditions evaluated to true. The calculate function did nothing, returned nothing.

Keyword Arguments - Invoking Functions

[Video description begins] *Topic title: Keyword Arguments- Invoking Functions. Your host for this session is Janani Ravi*. [Video description ends]

So far all of the input arguments that we've been working with in our custom functions have been positional arguments. These are required arguments and the order in which you specify those arguments when you invoke your function is important.

[Video description begins] *A Jupyter notebook named PositionalAndKeywordArguments displays. In code cell 1, the host types the following code: line 1: def introduction (name, city): line 2: print ("Hello, my name is", name) line 3: print ("I live and work in", city)*. [Video description ends]

Let's see what exactly that means. Here is our introduction function. This is a function that we've seen before, it takes two input arguments, name and city. Name is the first argument, its position is one. City is the second argument, its position is two.

And we printout to screen within this function, Hello my name is whatever name you specified, I live and work in this city. Let's invoke the introduction function, and we'll specify no values for

[Video description begins] *In code cell 2, she types the following code: introduction ( )*. [Video description ends]

the input arguments. We've tried this before, and we know that this doesn't work, this is an error. You can see that the TypeError here specifies very clearly that introduction has two required positional arguments, name and city.

And both of these are missing in our invocation. They're positional arguments because their values are assigned based on position. The first value goes to the first argument name, second value to the second argument city. And we'll see this in just a bit. Let's here invoke the introduction function and pass in just one value that will be assigned to the name variable, the first input argument.

[Video description begins] *In code cell 3, she types the following code: introduction ("Rose").* [Video description ends]

This invocation also results in a TypeError, because introduction has two required positional arguments. We just specified the first one that was assigned to the name variable. We haven't specified a value for the city input argument. The right way to invoke the introduction function is to specify values for both positional arguments, Rose from San Jose.

[Video description begins] *In code cell 4, she types the following code: introduction ("Rose", "San Jose").* [Video description ends]

This invocation will work. Hello, my name is Rose, I live and work in San Jose. Let's see another example here with positional arguments, the values of all of which have to be specified.

[Video description begins] *In code cell 5, she types the following code: line 1: def sum_of (num_1, num_2, num_3): line 2: return (num_1 + num_2 +num_3) line 3: sum_of (25, 34).* [Video description ends]

Here is a function called some of, it takes in three input arguments, num 1, num 2, num 3, and returns the sum of these numbers. There are three input arguments here. If I try to invoke the sum of function with just two arguments specified, that will be an error.

So sum of 25 plus 34, even though it seems meaningful, is not really allowed because of the way our custom function has been defined. You can see that the TypeError here says, missing this positional argument num_3. 25 was assigned to num 1, 34 to num 2 in the second position, we don't have a value for the third positional argument.

I'm going to define one more function here called total_score. This takes in four positional arguments, the score for math, physics, chemistry, and biology for a particular student.

[Video description begins] *In code cell 6, she types the following code: line 1: def total_score (math, physics, chemistry, biology): line 2: print ("Math:", math, "Physics:", physics, "Chemistry:", chemistry, "Biology:", biology) line 3: return math + physics + chemistry + biology.* [Video description ends]

I'm going to print out the original scores here for each of these subjects. And I'm going to return the sum of all of these scores, that is the return value from this function. Let's take a look at one way in which we can invoke the total_score function, call total_score and specify all numeric values.

[Video description begins] *In code cell 7, she types the following code: total_score (100, 56, 34, 99).* [Video description ends]

Now this kind of invocation is rather confusing, because you don't know what score is associated with what subject. Also you have to make sure that you specify the scores in the right order.

Here, 100 is the score associated with math, 56 with physics, 34 with chemistry, and 99 with biology. But, what if you mix these up? Based on what your function does, that can be a problem and there'll be no way to detect this problem.

So, let's invoke this function here, and you can see the result here on screen. This function works just fine. But we've seen that it's harder and harder to work with positional arguments when we have huge numbers of arguments and arguments are very similar.

And this is exactly the reason why Python offers support for keyword arguments. When you invoke a function using keyword arguments, you can specify values for input arguments, using the argument's name.

Observe this invocation of the total score function here, here we've used keyword arguments. I've used the name of the input argument to specify values for those input arguments.

[Video description begins] *In code cell 8, she types the following code: total_score (math=100, physics=56, chemistry=34, biology=99)*. [Video description ends]

The math score is 100, physics is 56, chemistry is 34, biology is 99. In this way, you know very clearly what value is associated with what variable and what input argument. Somebody reading this code or maintaining this code will find things simple as well.

The result of invoking this function is exactly the same as before, it's only the way we invoke the function that is different. We've used keyword arguments, instead of using the positions of arguments to specify values.

When you use keyword arguments, Python does a bunch of error checking as well. For example, if you try to specify a keyword that was not in the original argument list, that will be an error.

[Video description begins] *In code cell 9, she types the following code: total_score (math=100, physics=56, chem=34, biology=99)*. [Video description ends]

Observe the keyword chem here, we'll actually use the term chemistry. There is no keyword argument called chem, and that is clearly an error here in this function invocation.

So when you use keyword arguments while invoking functions in Python, make sure that you get your variable names correct, because Python will throw an error if you don't. And you can check that this is indeed the case with another example.

We will invoke the total score function and specify the first argument as ma. There is no input argument named ma in our function definition. That is clearly an error.

[Video description begins] *In code cell 10, she types the following code: total_score (ma=100, phy=56, chemistry=34, biology=99)*. [Video description ends]

Typically, you'll see in Python that people tend to invoke functions by mixing up positional arguments as well as keyword arguments as in this example here. And this is perfectly valid.

[Video description begins] *In code cell 11, she types the following code: total_score (100, physics=56, chemistry=34, biology=99)*. [Video description ends]

Here I have invoked the total score function, I've specified the value for the first input argument, 100, without a corresponding keyword. The remaining arguments I've specified using keywords physics, chemistry, and biology.

Now Python will know that 100 is associated with a positional argument that is the first argument in our function definition, the math score. And you'll see that the value 100 is associated with math automatically. So positional arguments are assigned to the input arguments of our custom function first, and then keyword arguments are assigned. Let's take a look at another example.

Here I've invoked total score, and the first two arguments, 100 and 56, do not have keywords associated with it. 100 is associated with math, 56 with physics. We've assigned the values to the first two input arguments by treating them as positional arguments.

The remaining values for chemistry and biology scores, we've assigned using keyword arguments. And this is perfectly valid as you see from the result here on screen. 100 is the math score, 56 is the physics score. These are positional arguments, chemistry and biology are key word arguments. Let's try one more example. The first three are positional arguments, only the score for biology I've specified using a keyword argument.

[Video description begins] *In code cell 13, she types the following code: total_score (100, 56, 34, biology=99)*. [Video description ends]

These are examples of how we can mix and match positional and keyword arguments in a way that works. Now let's see what does not work. We'll continue to work with the total score custom function that we've defined. Here I have used a mix of positional and keyword arguments. The first argument, math score, is a keyword argument then 56 and chemistry and biology I've also specified using keyword arguments.

[Video description begins] *In code cell 14, she types the following code: total_score (math=100, 56, chemistry=34, biology=99)*. [Video description ends]

Well, Python does not like this. You can mix and match positional and keyword arguments only if all of your positional arguments are specified first. So specify all of your positional arguments, and then specify the keyword arguments, that works.

You can't specify keyword arguments and then include a positional argument somewhere in between. Let's see another invocation that will throw an error. I have invoked the total score function, I've specified math and physics as keyword arguments. And suddenly, chemistry, I specify as a positional argument. And that's not allowed as you can see from the error here.

[Video description begins] *In code cell 15, she types the following code: total_score (math=100, physics=56, 34, biology=99)*. [Video description ends]

Positional arguments cannot follow keyword arguments. Keyword arguments can follow positional arguments. One more example of an error, the first three arguments are keyword arguments, the last argument is positional, and that is wrong.

[Video description begins] *In code cell 16, she types the following code: total_score (math=100, physics=56, chemistry=34, 99)*. [Video description ends]

Keyword Arguments - Nuances

[Video description begins] *Topic title: Keyword Arguments- Nuances. Your host for this session is Janani Ravi*. [Video description ends]

Using keyword arguments to invoke Python functions are actually very, very useful in a variety of different ways.

[Video description begins] *The PositionalAndKeywordArguments notebook displays. In code cell 17, the host types the following code: total_score (biology=99, math=100, chemistry=34, physics=56)*. [Video description ends]

Another useful thing, you don't have to specify your arguments in order. Notice our invocation of the total_score function here. Observe that the first input argument that I've specified is for the biology score. Then math, then chemistry, then physics.

Whereas in our original function definition, we had specified the score for math as the first input argument, then physics, chemistry, and then biology. But while invoking the total score function, I have specified the biology score first. And that's totally fine.

That's the beauty of using keyword arguments to invoke a function. And you can see that we get the results that we expect. When you invoke functions using keyword arguments, the order of your arguments does not matter. Order is important only when you're using positional arguments to invoke a function.

[Video description begins] *In code cell 18, she types the following code: total_score (chemistry=34, biology=99, math=100, physics=56)*. [Video description ends]

Observe that I've mixed up the order here in this function invocation, the chemistry score is specified first, but the function works just fine. Your input arguments have unique names. Their variables have unique names, and you can't specify two values for the same input argument.

[Video description begins] *In code cell 19, she types the following code: total_score (34, biology=99, math=100, physics=56)*. [Video description ends]

Observe our total score invocation here. We've used a positional argument at the beginning which is allowed, 34 is the score assigned to math. But in the same function invocation, we've explicitly used a keyword argument to assign the score for math.

We've specified math is equal to 100 and we haven't specified any value for chemistry. Now, this is an error. Total score got multiple values for the argument math, and it did not get any value for chemistry. We specified a value for math using a positional argument, and once again, using a keyword argument and that's not allowed.

If you're working with a function that has many input arguments, you might, by mistake, specify a value for an argument more than once. Observe that we've specified the value for the physics score twice here. And that's not allowed and you can see that we have a syntax error, keyword argument repeated.

[Video description begins] *In code cell 20, she types the following code: total_score (physics=34, biology=99, chemistry=100, physics=56)*. [Video description ends]

I think we're now ready to move on from the total score custom function. Let's look at another custom function here called print student details and it has a total of six input arguments.

[Video description begins] *In code cell 21, she types the following code:line 1: def print_student_detail (name, school, math, physics, chemistry, biology): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school) line 5: print ("Score:", total)*. [Video description ends]

The name of the student, the name of the school he or she studies in, and scores for math, physics, chemistry and biology in that order. Within this function, we calculate the total score, print out the name of the student and school, and the total score.

When you're invoking a function of this form, it's quite common to use positional arguments to specify the earlier values where there are just two string values to be specified, and keyword arguments for the scores for math, physics, chemistry, etc. Observe this invocation here. This is perfectly valid, and you'll see that this function executes successfully.

[Video description begins] *In code cell 22, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64, biology= 99)*. [Video description ends]

Now we've seen the drawback when we use positional arguments. If you mix up your positions of the arguments that you specify, you'll get a wrong result. Here I've specified Columbia as the name of the student and Alice as the name of the school, which is clearly wrong.

[Video description begins] *In code cell 23, she types the following code: print_student_detail ("Columbia", "Alice", math=100, physics= 76, chemistry= 64, biology= 99)*. [Video description ends]

And in order to prevent such wrong method invocations is exactly why you need to use keyword arguments rather than positional arguments. Name is equal to Alice, school is Columbia. All keyword arguments are used here. This is an example of a good invocation.

[Video description begins] *In code cell 24, she types the following code: print_student_detail (name="Alice", school="Columbia", math=100, physics= 76, chemistry= 64, biology= 99)*. [Video description ends]

It's very common to use keyword arguments for specific input arguments with built in functions in Python. Here is the sorted function which sorts the element of a list. When we only want to specify the list that we want to sort, we pass in the num_list using positional arguments.

[Video description begins] *In code cell 25, she types the following code: line 1: num_list =[3, 6, 1, 9, 5, 7] line 2: sorted (num_list)*. [Video description ends]

The output of the sorted function here is this num_list sorted in ascending order. It turns out that you can invoke the sorted function by specifying a value for the reverse input argument. I've specified it as a keyword argument here, reverse is equal to false and this is the default value for reverse. Reverse is by default false.

[Video description begins] *In code cell 26, she types the following code: sorted (num_list, reverse= False)*. [Video description ends]

We'll talk more about default arguments in just a bit, but you should know that lists are sorted in the ascending order by default, which means the reverse input argument is false by default. And observe how I've used keyword arguments to specify the value for reverse here. Now if you want the list sorted in the ascending order, you'll specify reverse is equal to true.

[Video description begins] *In code cell 27, she types the following code: sorted (num_list, reverse= True)*. [Video description ends]

Observe how using a keyword argument to specify the value for the reverse input argument makes it very clear how we plan to sort this num_list in the reverse order, in the descending order. Let's take another example, the built in print function.

[Video description begins] *In code cell 28, she types the following code: print ("Alice", "Peter", "Greg")*. [Video description ends]

We'll print out Alice, Peter and Greg to screen, and it will be displayed as you see here. Now what if I want Alice, Peter and Greg to be printed out but using a different separator? Well, I can use a keyword argument for the specification.

The default separator in the print function is the empty string, but here I've specified the pipe as a separator and you can see that Alice, Peter and Greg have been printed out using the pipe to separate them.

[Video description begins] *In code cell 29, she types the following code: print ("Alice", "Peter", "Greg", sep = " | ")*. [Video description ends]

You can also use the end input argument to print out something at the end of the rest of your print statements. The default end specification is the /n which is a new line. You can also have these asterices to mark the end of your print.

[Video description begins] *In code cell 30, she types the following code: print ("Alice", "Peter", "Greg", sep = " | ", end = "*****")*. [Video description ends]

What we just discussed are examples of built-in functions where we used keyword arguments in order to make things clear about what exactly we were specifying as an input to that function. Whether it's a separator, the end of our print statement, the reverse, true, false, for sorted, etc.

Default Arguments

[Video description begins] *Topic title: Default Arguments. Your host for this session is Janani Ravi*. [Video description ends]

In the real world, you'll find that functions often have many, many arguments that you need to pass in. But all of those arguments may not be required. For some of these arguments, you might have meaningful default values.

[Video description begins] *A blank notebook named DefaultArguments displays*. [Video description ends]

And this is exactly what we'll study here now, how we can specify meaningful default values for input arguments. Let's take a look at the print_student_details function that we've seen before.

[Video description begins] *In code cell 1, the host types the following code: line 1: def print_student_detail (name, school, math, physics, chemistry, biology, enrolled): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Score:", total)*. [Video description ends]

I've modified this function a little bit. It takes in seven input arguments, the name of the student, the name of the school, scores in math, physics, chemistry, and biology, and whether the student has enrolled in the school or not.

Within the body of the function, we calculate the total score for the student, print out the name, school and whether the student is enrolled or not, as well as the total score.

A very simple function, but with many input arguments. In order to invoke this function to get it to perform something useful, you have to specify values for all of these input arguments, whether using positional arguments or keyword arguments. Here are the details for Alice who goes to Columbia.

[Video description begins] *In code cell 2, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64, biology= 99, enrolled=True)*. [Video description ends]

Because of the number of arguments that this function has, invoking this function is actually quite complex. And this is the result of our invocation, this was a successful invocation, everything went fine.

But you can see that invoking this required us to specify many input arguments. If you forget or miss out the value for even one of these input arguments, your function invocation will fail. As you can see here, I've forgotten to specify a value for the enrolled input argument and that's why I get this error.

[Video description begins] *In code cell 3, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64, biology= 99)*. [Video description ends]

It's possible for us to rewrite our print_student_details function so that enrolled is not a required argument, instead it has a meaningful default value. And that's exactly what I've done here. print_student_details we still require values for six input arguments. But observe that the enrolled input argument has a default value, by default enrolled=False.

[Video description begins] *In code cell 4, the host types the following code: line 1: def print_student_detail (name, school, math, physics, chemistry, biology, enrolled=False): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Score:", total)*. [Video description ends]

The default value for any of the input arguments to a function is specified as a part of the function definition. enrolled=False is specified in the arguments list.

When your argument has a default value, you can still invoke the function and specify a different value for that argument. Observe that we've invoked the print_student_details function here, but I have specified that enrolled=True.

[Video description begins] *In code cell 5, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64, biology= 99, enrolled=True)*. [Video description ends]

With this function invocation, what you're essentially telling Python is, Alice is indeed enrolled in Columbia, enrolled=True. Do not take the default value for enrolled, take the value that I have specified, she is indeed enrolled in Columbia.

And you can see in the result that within the function, enrolled is indeed true. You can invoke the same function and explicitly specify enrolled=False as well, this is also allowed. Just because enrolled=False is the default value for enrolled, you can still specify it when you explicitly invoke the function. You can see that Alice is not enrolled in Columbia here.

[Video description begins] *In code cell 6, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64, biology= 99, enrolled=False).* [Video description ends]

But where default values for function arguments are useful is where they reduce the burden on the invoker of the function. I'll invoke print_student_details here, but I haven't specified a value for the enrolled function. Now normally, this would have been an error.

We tried the same invocation before and that was a failure, but this time, observe that the function is invoked successfully and executes successfully.

[Video description begins] *In code cell 7, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64, biology= 99).* [Video description ends]

Also observe the value of enrolled, enrolled=False. And this value was received within the function using the default value specification in the function definition. Then we set up the print_student_details function.

We had specified the default value for enrolled which is why you can invoke the function without specifying a value for the enrolled variable. I'm now going to redefine the print_student_details function, but this time I'll specify default values for two of the input arguments.

The default score for biology is 50 and enrolled is False by default.

[Video description begins] *In code cell 8, the host types the following code: line 1: def print_student_detail (name, school, math, physics, chemistry, biology=50, enrolled=False): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Biology:", biology) line 6: print ("Score:", total).* [Video description ends]

When arguments have default values, it essentially means that you don't have to specify values for these input arguments while invoking the function. Here is an invocation of print_student_details, here I've specified a value for biology score, biology=99, but I haven't specified a value for the enrolled variable.

[Video description begins] *In code cell 9, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64, biology= 99).* [Video description ends]

The biology score for this function invocation will be equal to 99 but the student will not be enrolled. The value for the enrolled variable comes from the default value, by default, enrolled=False.

Because the biology score input argument as well as enrolled both have default values, you can invoke print_student_details and specify values for all input arguments except biology and enrolled. And this invocation is perfectly correct.

You can see here from the statements printed out to screen that enrolled=False, that is the default value, and the biology score is 50, once again the default value.

[Video description begins] *In code cell 10, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry= 64).* [Video description ends]

If you try to invoke the print_student_details function and leave out the value of an input argument which does not have a default value of its own, well, that's an error. Here we haven't specified a score for chemistry, and that's why you see this error here, we are missing one required positional argument.

[Video description begins] *In code cell 11, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76).* [Video description ends]

I'm now going to redefine the print_student_details function and specify a default value for the chemistry score as well. The default value here is once again 50.

[Video description begins] *In code cell 12, the host types the following code: line 1: def print_student_detail (name, school, math, physics, chemistry=50, biology=50, enrolled=False): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Chemistry:", chemistry)line 6: print ("Biology:", biology) line 7: print ("Score:", total).* [Video description ends]

Now three of our input arguments have default values chemistry, biology, and enrolled. Let's invoke this function, and this time, I have specified a value for the chemistry score, so that is the value that will be seen within the function. You can see that chemistry is equal to 64.

[Video description begins] *In code cell 13, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76, chemistry=64).* [Video description ends]

But while invoking the function if you leave out explicitly specifying values for those input arguments that have defaults, the defaults are what will be accepted within the function. You can see that enrolled is False, that is the default, chemistry is 50 and biology is 50.

[Video description begins] *In code cell 14, she types the following code: print_student_detail ("Alice", "Columbia", math=100, physics= 76).* [Video description ends]

Now, Python has a rule about arguments which have default values. If these exist, these need to be specified at the very end of the argument list which is passed into your function. Observe the input arguments of the print_student_details function here.

This is actually wrong, and in a minute I'll explain why. Observe that I've specified a default value for math, no default value for physics, and then default values for chemistry, biology, and enrolled.

[Video description begins] *In code cell 15, the host types the following code: line 1: def print_student_detail (name, school, math=50, physics, chemistry=50, biology=50, enrolled=False): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Chemistry:", chemistry)line 6: print ("Biology:", biology) line 7: print ("Score:", total).* [Video description ends]

When I try to define this function, Python gives me a syntax error, non-default argument follows default argument. So math has a default value, it is a default argument. And physics follows the math argument but with no default value and this is not allowed.

All of the arguments that have default values should be specified at the end of the arguments list. Let's take a look at another example that ought to make this very, very clear. Here is the print_student_details function, observe that I have specified a default value for school, school=Harvard.

I haven't specified default values for math and physics course. Chemistry, biology, and enrolled have default value specifications. Now this is, again, not allowed. We have a default argument school which is followed by math and physics, which are not default arguments, and Python doesn't allow this.

[Video description begins] *In code cell 16, the host types the following code: line 1: def print_student_detail (name, school="Harvard", math, physics, chemistry=50, biology=50, enrolled=False): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Chemistry:", chemistry)line 6: print ("Biology:", biology) line 7: print ("Score:", total)*. [Video description ends]

One more wrong function definition so that it's a 100% clear in your head, name="Mark", that is a default argument. Followed by school, math, and physics, which are non-default arguments which is why you see this error here.

[Video description begins] *In code cell 17, the host types the following code: line 1: def print_student_detail (name="Mark", school, math, physics, chemistry=50, biology=50, enrolled=False): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Chemistry:", chemistry)line 6: print ("Biology:", biology) line 7: print ("Score:", total)*. [Video description ends]

And finally, here is a function definition that works. You can specify default values for all of your input arguments. Observe that for the student Mark who studies at Harvard, we have default values for all of his scores, he's not enrolled yet.

[Video description begins] *In code cell 18, the host types the following code: line 1: def print_student_detail (name="Mark", school="Harvard", math=50, physics=50, chemistry=50, biology=50, enrolled=False): line 2: total = math +physics+chemistry+biology line 3: print ("Name: ", name) line 4: print ("School: ", school, "Enrolled:", enrolled) line 5: print ("Math:", math)line 6: print ("Physics:", physics) line 7: print ("Chemistry:", chemistry)line 8: print ("Biology:", biology) line 9: print ("Score:", total)*. [Video description ends]

You can invoke such a function by specifying no input arguments. Simply call print_student_details, we haven't specified any values within parenthesis, this works. All of the input arguments take on their default values.

[Video description begins] *In code cell 19, the host types the following code: print_student_detail ( )*. [Video description ends]

And now, with this function, you can specify values for as many arguments as you please. I pass in Alice, name will be Alice, all of the other arguments will have their default values.

[Video description begins] *In code cell 20, the host types the following code: print_student_detail ("Alice")*. [Video description ends]

I specify Alice and Columbia, name and school will be Alice and Columbia respectively, and all of the other input arguments have their default values. All scores are 50, the student is not enrolled.

When all your input arguments have default values, you have many ways in which you can invoke this function. Here, I have specified positional as well as keyword arguments. Alice, Columbia, and 99 correspond to name of school and the math score, the biology score is 66 and the student is enrolled.

The remaining values all are from the default specification, the physics score and the chemistry score.

[Video description begins] *In code cell 21, the host types the following code: print_student_detail ("Alice", "Columbia", 99, biology=66, enrolled=True)*. [Video description ends]

Variable Length Arguments - *args Variable

[Video description begins] *Topic title: Variable Length Arguments- *args Variable. Your host for this session is Janani Ravi*. [Video description ends]

So far, all of the functions that we've worked with have specified a fixed number of arguments to receive.

[Video description begins] *A blank notebook named VariableLengthArguments displays*. [Video description ends]

You can pass these arguments in either as positional arguments or keyword arguments and that gives you some flexibility. Additional flexibility is what you get using default arguments. But what if you could have functions which accepts any number of arguments?

How would you set that up in Python? That's exactly what we'll cover here in this demo. Before we define our own custom function, let's consider a built-in function that we are familiar with, the print function.

[Video description begins] *In code cell 1, the host types the following code: print ( )*. [Video description ends]

Now you might have invoked the print function like this. Here, we passed in no input arguments to the print statement. In such conditions, print simply prints out a new line to screen, there's nothing printed out. But you've also used the print statement by passing in just a single string, this prints Bob to screen.

[Video description begins] *In code cell 2, she types the following code: print ("Bob")*. [Video description ends]

We've also used print statements where we've passed in multiple input arguments, two strings, Bob and Peter are printed out to screen.

[Video description begins] *In code cell 3, she types the following code: print ("Bob", "Peter")*. [Video description ends]

We've used print with three arguments, four arguments, you name it, all of these work.

[Video description begins] *In code cell 4, she types the following code: print ("Greg", "Claudia", "Bob")*. [Video description ends]

So clearly, the print built-in function accepts variable length arguments. How does this happen?

[Video description begins] *In code cell 5, she types the following code: print ("Greg", "Claudia", "Bob", "Peter")*. [Video description ends]

Well, that's what we're about to find out. Let's try this out. We have our own print function called print_fn and it takes in one input argument, and we print that one input argument out to screen.

[Video description begins] *In code cell 6, she types the following code: line 1: def print_fn (string_1) : line 2: print (string_1)*. [Video description ends]

The single argument to this custom print function is require. So if you try and invoke this custom print, without specifying any arguments, that is an error, you can see that we are missing one required positional argument.

[Video description begins] *In code cell 7, she types the following code: print_fn ( )*. [Video description ends]

All right, let's modify this. I'm going to have the single argument have a default value which is equal to /n which is the new line character.

[Video description begins] *In code cell 8, she types the following code: line 1: def print_fn (string_1= '\n') : line 2: print (string_1)*. [Video description ends]

Now that the single input argument has a default value, you can invoke the print function with no arguments at all. And this will print out a new line on to screen.

[Video description begins] *In code cell 9, she types the following code: print_fn ( )*. [Video description ends]

We've managed to mimic some of the behavior of the built-in print function using this custom function. You can also invoke this custom function with a single argument.

[Video description begins] *In code cell 10, she types the following code: print_fn ("Bob").* [Video description ends]

But what if you were to try and print out two strings to screen, Bob and Peter? Well, this print function will not work. Because we have just one positional argument defined in this function.

[Video description begins] *In code cell 11, she types the following code: print_fn ("Bob", "Peter").* [Video description ends]

Using all of the concepts that we've studied so far about default arguments and so on, here is another function that I've set up which is a little tricky. We have two input arguments. The default value for the first input argument is the new line character.

And for the second, it's the empty string. So if I invoke the print function with no input argument,

[Video description begins] *In code cell 12, she types the following code: line 1: def print_fn (string_1= '\n', string_2= " ") : line 2: print (string_1, string_2).* [Video description ends]

it'll simply print the new line to screen.

[Video description begins] *In code cell 13, she types the following code: print_fn ( ).* [Video description ends]

I can invoke it with one input argument and it'll simply print Bob out to screen here in this example. I can also invoke the print function with two input arguments.

[Video description begins] *In code cell 14, she types the following code: print_fn ("Bob").* [Video description ends]

But if you go beyond two, our custom print function will not support this.

[Video description begins] *In code cell 15, she types the following code: print_fn ("Bob", "Peter").* [Video description ends]

If you try and print three strings, Greg, Claudia, and Bob, for example, our custom print function will simply say, the print function takes from 0 to 2 positional arguments but 3 were given.

[Video description begins] *In code cell 16, she types the following code: print_fn ("Greg", "Claudia", "Bob").* [Video description ends]

So how do we get this print function to accept any number of arguments like the built-in print function does? Well, there is a special concept for exactly this.

[Video description begins] *In code cell 17, she types the following code: line 1: def print_fn (*args): line 2: args_type= type (args) line 3: print (args_type) line 4: print (args).* [Video description ends]

If you want your custom function to accept variable numbers of arguments, you'll specify it as you see here on screen. Here is a print function. Observe how I've specified just the single input argument args here. It's preceded by the asterisk.

This is the syntax Python uses to indicate that you have a function that can accept variable numbers of arguments, an asterisk and the name of the variable. Typically the name of the variable is args, but you can specify other names as well.

Before we do anything with the input arguments that are passed in, let's examine what exactly this args is all about. I'm going to find the type of the args argument that this function receives and print out the type to screen, I'll also print out args itself. Now let's see how this works. I'm going to invoke the print function and specify no input argument at all.

[Video description begins] *In code cell 18, she types the following code: print_fn ( ).* [Video description ends]

You'll find that the invocation is successful, it does not throw an error. And you can also see that there are two bits of information printed out to screen. The first is the args type and you can see that the args argument that this print function receives is a tuple, the args argument is a tuple.

And you can see that right now, it's an empty tuple because we invoke the print_fn() without specifying any input arguments. Now this gives us a crucial bit of understanding. When you have a custom function, received *args as its input argument.

Within that function, the variable numbers of arguments that are passed in are passed in in the tuple data type. The fact that all of the inputs are received in the form of a tuple is what gives this function the flexibility to accept variable numbers of arguments. Let's invoke the same print function and pass in one input argument, the string Bob.

[Video description begins] *In code cell 19, she types the following code: print_fn ("Bob").* [Video description ends]

The data type of the args input argument is of class tuple. And you can see that the tuple contains exactly one field, the string Bob that we had passed in. Now let's invoke the print function with two input arguments. I'm going to try and print out Bob as well as Peter.

[Video description begins] *In code cell 20, she types the following code: print_fn ("Bob", "Peter").* [Video description ends]

Once again, you can see that the args data type is a tuple. It's of class tuple and you can see that the tuple contains two fields. Field one is a Bob, that is our first input argument. Field two is Peter, that is our second input argument.

You can try this out on your own. Invoke this print function with as many input arguments as you want and you'll find that all of these will be accepted. All of these will be wrapped into the args tuple when the print function receives these input arguments.

[Video description begins] *In code cell 21, she types the following code: print_fn ("Greg", "Claudia", "Bob").* [Video description ends]

The presence of the asterisk indicates to Python that there can be any number of arguments. Just pack them all into the tuple format and pass a single tuple into this Python function.

[Video description begins] *In code cell 22, she types the following code: print_fn ("Greg", "Claudia", "Bob", "Alice", "Steve", "Jim").* [Video description ends]

What if all of the inputs that you want to pass into a function are already available in the list or tuple format? Here is a list of names, and these are the list of names that I want to print out using the print function. So I invoke the print function and pass in just names_list.

[Video description begins] *In code cell 23, she types the following code: line 1: names_list = ["Greg", "Claudia", "Bob", "Alice", "Steve", "Jim"] line 2: print_fn(names_list).* [Video description ends]

When your input argument is specified as the list itself, the list is considered to be a single entity, which is then packed into a tuple. You can see that the print function receives the list, but the list is the first field in a tuple.

Python has considered the entire list as a single element, and it's the first field in the tuple that is passed in. But what if you want the individual elements that make up the names_list to be separate input arguments to your print function?

[Video description begins] *In code cell 24, she types the following code: print_fn(*names_list)*. [Video description ends]

Well, you will ask Python to unpack the names_list and then pass this unpacked list into the print function. Once inside the print function, Python will repack the individual elements of the names_list as individual elements of the tuple that is passed into the print function, the args tuple. Here, you can see that the individual elements in the names_list, all of the names are different fields in the tuple.

Variable Length Arguments - Combinations

[Video description begins] *Topic title: Variable Length Arguments- Combinations. Your host for this session is Janani Ravi*. [Video description ends]

The use of variable length arguments gives us a lot of flexibility in defining our custom functions.

[Video description begins] *The VariableLengthArguments notebook displays. In code cell 25, she types the following code: line 1: def students_in_college (college, city, *students): line 2: print ("College:", college) line 3: print ("City:", city) line 4: print ("Students:", students)*. [Video description ends]

It's possible for you to have a custom function which has a few required arguments. And there are additional arguments that can be variable length arguments. Take a look at this function here on the screen. Students_in_college, this has two required arguments.

The name of the college and the city in which that college is located. The third input argument here, *students, is a variable length argument, which means that you can specify the names of zero or more students who study in this particular college.

College and City are positional arguments that are required. They don't have default values, but you may or may not include the names of students when you invoke this function. Let's see an example. Let's invoke students in college and specify no input arguments at all.

[Video description begins] *In code cell 26, she types the following code: students_in_college ( )*. [Video description ends]

This is clearly an error because we have two required positional arguments whose values were not specified. You can see that the error makes this very clear. We are missing two required positional arguments, college as well as city.

There is no mention of the variable length argument students here because that's not required. Let's now invoke this function by specifying a value for just the first required positional argument, the name of the college, that is Columbia.

[Video description begins] *In code cell 27, she types the following code: students_in_college ("Columbia")*. [Video description ends]

Once again, this is an error because we haven't specified a value for city, which is also a required positional argument. If I invoke the students_in_college function and specify the name of the college as well as the city where the college is located, well, this invocation works perfectly fine.

[Video description begins] *In code cell 28, she types the following code: students_in_college ("Columbia", "New York")*. [Video description ends]

The students input argument, which is a variable length argument, and you can choose to specify no values for it at all. Observe that the first value that you specified, Columbia, is assigned to college. The second value is assigned to city, both of these are required.

And the remaining values, here there are none, will be assigned to the students variable. Here is another invocation of the students_in_college function. And here we have specified that the student Bob studies in Columbia based in New York City. Observe that the first two input arguments correspond to college and city.

[Video description begins] *In code cell 29, she types the following code: students_in_college ("Columbia", "New York", "Bob")*. [Video description ends]

And the remaining arguments may be assigned to the students variable. You can see from the results printed out to screen that all of the remaining arguments after college and city have been assigned values, are assigned to students and they make up a tuple.

Let's confirm what I just said. I'm going to invoke the students_in_college function once again, pass in Columbia, New York and pass in two student names, Peter as well as Julie.

[Video description begins] *In code cell 30, she types the following code: students_in_college ("Columbia", "New York", "Peter", "Julie")*. [Video description ends]

You can see that leaving aside the first two values that are assigned to the college and city, the remaining values are all assigned to the students input argument, which is a variable length argument. And Python has packed all of the input values in the form of a tuple.

You can see that the students, Peter and Julie, are available as a tuple within our function. You can practice invoking the students_in_college once more and this time we have specified three students, Peter, Julie, and Trista, all of whom study in Columbia in New York.

And you can see that the students input argument has been packed into a tuple with three fields with the names of the three students. When you use keyword arguments to specify the values for the city and college input argument, you'll find that Python doesn't really like this. Now, it may not always be intuitive what Python can and cannot understand.

[Video description begins] *In code cell 31, she types the following code: students_in_college (city="New York", college= "Columbia" , "Peter", "Julie")*. [Video description ends]

So often you'll have to rely on the errors that you receive to see whether something is allowed or not. The error that we get here is that we've specified city and college as keyword arguments and then specified variable length arguments without a keyword.

You know that positional arguments cannot follow keyword arguments and that's the error that we get here. Python has a few other constraints on variable length arguments defined as a part of a function.

Observe this function here and observe that students, our variable length argument, is the first input argument to this function, followed by college and city.

[Video description begins] *In code cell 32, she types the following code: line 1: def students_in_college (*students, college, city): line 2: print ("College:", college) line 3: print ("City:", city) line 4: print ("Students:", students)*. [Video description ends]

When you set up your custom function in this manner, you'll find that there are a few constraints in how you can invoke this function. I've invoked students_in_college and I specify the name of the college first, then city and then student names.

[Video description begins] *In code cell 34, she types the following code: students_in_college ("Columbia", "New York", "Bob").* [Video description ends]

Python throws an error here because it believes that Columbia, New York, and Bob are all part of the variable length argument students. So it says, I didn't find values for college and city, those are required, and that's why I've failed.

In order to invoke this function successfully, you need to call students_in_college pass in Bob, that is the name of the student. But you need to pass in college and city values using keyword arguments as you see here on screen.

[Video description begins] *In code cell 35, she types the following code: students_in_college ("Bob", college= "Columbia", city="New York").* [Video description ends]

And this invocation will work fine because we've explicitly specified college and city as keyword arguments.

The remaining arguments will all be assigned to the students variable. And you'll find that this technique works even when you have multiple student names in your variable arguments list. Bob and Peter are the two students, the college is Columbia, the city is New York and this works fine as you can see.

[Video description begins] *In code cell 36, she types the following code: students_in_college ("Bob", "Peter", college= "Columbia", city="New York").* [Video description ends]

You can also mix and match variable length arguments, as well as input arguments that have default values. Here the students and college function has been defined, the variable argument is the students variable.

We have the college, but we've said that we're only interested in colleges in the city of Boston, that is our default argument.

[Video description begins] *In code cell 37, she types the following code: line 1: def students_in_college (*students, college, city="Boston"): line 2: print ("College:", college) line 3: print ("City:", city) line 4: print ("Students:", students).* [Video description ends]

If you try to invoke this function without specifying college as a keyword argument, you'll find that this results in an error. Students_in_college missing 1 required keyword-only argument.

[Video description begins] *In code cell 38, she types the following code: students_in_college ("Bob").* [Video description ends]

Let's say you invoke this function by specifying Bob as well as Harvard. You expect Harvard to be assigned to the college input argument. Well, that's not possible unless you use a keyword specification.

[Video description begins] *In code cell 39, she types the following code: students_in_college ("Bob", "Harvard").* [Video description ends]

So the correct way to invoke this function is to invoke it by specifying college equal to Harvard or any other college.

[Video description begins] *In code cell 40, she types the following code: students_in_college ("Bob", college="Harvard").* [Video description ends]

Basically, when your variable length argument, such as the students argument, is the first argument in the list, all of the remaining arguments become keyword only arguments.

Variable Length Arguments - **kwargs Keyword

[Video description begins] *Topic title: Variable Length Arguments- **kwargs Keyword. Your host for this session is Janani Ravi*. [Video description ends]

Python offers another way that you can specify variable-length arguments as an input to a function.

[Video description begins] *The VariableLengthArguments notebook opens*. [Video description ends]

And this, in fact, is a more intuitive syntax. You can use keyword arguments instead of just a tuple of variable-length arguments.

If you want your variable-length arguments to be passed in in the form of keywords, you'll specify the double asterisk in the front of your input argument, as you see here on screen. And a common variable name for keyword arguments is kwargs.

When you use the double asterisk, Python under the hood packs the variable-length arguments into a dictionary, not a tuple. This is a dictionary of key-value pairs, where the keys are the names of your variables, and the values are the values that you specify.

This will become clearer when you see how this function works. This is a student_details function. And it takes in a variable-length argument in the form of keywords. I'm going to print out the type of kwargs within this function, and print kwargs to screen as well.

[Video description begins] *In code cell 42, she types the following code: line 1: def student_details (**kwargs): line 2: print (types (kwargs)) line 3: print (kwargs)*. [Video description ends]

Because kwargs is a variable-length argument, you can invoke the student_details function and specify no values at all, and this is perfectly fine. Observe that the type of the kwargs argument is a dictionary.

[Video description begins] *In code cell 43, she types the following code: student_details ( )*. [Video description ends]

And this dictionary in this invocation is empty, we haven't specified any input values. I'll now invoke the student_details function with just one input keyword argument, name="John".

[Video description begins] *In code cell 44, she types the following code: student_details (name "John")*. [Video description ends]

Now you can specify any keyword arguments here at the input, and that's totally fine. Python will pack these arguments into a dictionary format. And you can see that kwargs is a dictionary with just one key-value pair, name is the key, John is the value.

Here is another invocation of the student_details function. This time, I've specified two keyword arguments, name="John", and age=18.

[Video description begins] *In code cell 45, she types the following code: student_details (name "John", age=18)*. [Video description ends]

These will be packed into the kwargs dictionary, and you can see that the dictionary now contains two key-value pairs, name as well as age.

And this is how variable-length keyword arguments work. You can specify keyword arguments in the form of name-value pairs, and these will be packed into a dictionary by Python. And kwargs will be a dictionary with all of the arguments that you passed in. Kwargs is the usual variable name for variable-length keyword arguments, but you can name it anything else as well.

[Video description begins] *In code cell 46, she types the following code: student_details (name "John", age=18, college= "Boston University").* [Video description ends]

For example, here our variable-length keyword argument is called details. Because it is a dictionary within a function, you can iterate over the items in detail using a for loop. Within this function, we're using a for loop to print out the keys and values in the details dictionary.

[Video description begins] *In code cell 47, she types the following code: line 1: def student_details (**details): line 2: for key, value in details.items ( ): line 3: print (key, value).* [Video description ends]

Let's invoke the student_details function with name, age, and college. And you can see that the key-value pairs are printed out to screen.

[Video description begins] *In code cell 48, she types the following code: student_details (name "John", age=18, college= "Boston University").* [Video description ends]

Let's say the values that you want to use to invoke the student_details function are already present in the form of a dictionary. Here is the details dictionary, with the name, age, and college for John.

[Video description begins] *In code cell 49, she types the following code: details_dictionary= ('name': 'John', 'age', 18, 'college': 'Boston University').* [Video description ends]

If you try to invoke the student_details function, and pass in the dictionary directly, that will not work. Because it's expecting keyword arguments, and details dictionary is passed in as a positional argument. And that's what this error here tells us.

[Video description begins] *In code cell 50, she types the following code: student_details (details_dictionary).* [Video description ends]

Instead, if you invoke the student_details function using the double asterisk before the details dictionary, this will tell Python to unpack the key-value pairs in the details dictionary. And pass them in in the form of keyword arguments to the student_details function.

[Video description begins] *In code cell 51, she types the following code: student_details (**details_dictionary).* [Video description ends]

So within the print function, the keyword arguments are once again available in the dictionary format. Now that we know that when we use the double asterisk with our input argument, we get keyword arguments in the form of a dictionary.

You can perform all kinds of dictionary-related operations with your input argument. In this implementation of the student_details function, I'm using the if conditional statement to check whether a particular key is present in the input dictionary. To see whether the value for name, age, or college has been passed in. If yes, I'm printing out that information to screen.

[Video description begins] *In code cell 52, she types the following code: student_details (**details_dictionary): line 2: if 'name' in details : line 3: print ("Name:", details ['name']) line 4: if 'age' in details : line 5: print ("Age:", details ['age']) line 7: if 'college' in details : line 8: print ("College:", details ['college']).* [Video description ends]

While invoking this function, if we only pass in a value for the name keyword argument, only name will be printed out to screen. We check whether name is present in the details dictionary, yes it is, we print out name. Age and college are not present.

[Video description begins] *In code cell 53, she types the following code: student_details (name "John").* [Video description ends]

If we invoke the function by specifying values for name as well as college, both name and college will be printed out to screen. Age is not available, it's not printed out.

[Video description begins] *In code cell 54, she types the following code: student_details (name "John", college= "Boston University").* [Video description ends]

Let's invoke this function once again. This time I pass in keyword arguments for name, college, and age, in any order. And all three bits of information are printed to screen.

[Video description begins] *In code cell 55, she types the following code: student_details (name "John", college= "Boston University", age=21).* [Video description ends]

One last word on variable-length arguments before we move on to other topics. You can have the same function, except variable-length arguments as positional arguments, as well as keyword arguments, as you see in this example here.

[Video description begins] *In code cell 56, she types the following code: line 1: def students_in_college (*student_names, **college_details): line 2: print ("Students--") line 3: for a in student_names: line 4: print (s) line 5: print ( ) line 6: print ("College Details--") line 7: for key, value in college_details.items ( ): line 8: print (key, value).* [Video description ends]

The student_names argument is preceded by a single asterisk, indicating that it's a positional variable-length argument. College_details has the double asterisk, indicating that it's a keyword variable-length argument.

Here is one way that you could invoke this function. Students_in_college "Allison", "Bob", and "Charlie" will be packed up into a tuple and assigned to student_names. The name is equal to Stanford and city is Palo Alto will be assigned to college_details. It'll be packed into the dictionary format.

[Video description begins] *In code cell 57, she types the following code: line 1: def students_in_college ("Alison", "Bob", "Charlie", name ="Stanford", city= "Palo Alto").* [Video description ends]

And when you invoke this function and run this code, that's exactly what you'll see is printed out from within the body of the function.

Exercise: Introduction to Functions in Python

[Video description begins] *Topic title: Exercise: Introduction to Functions in Python. Your host for this session is Janani Ravi.* [Video description ends]

Now that you're fairly comfortable with functions in Python. Let's get moving with some exercises that will test your recollection and your understanding. In this exercise, you will first start by describing what functions are and how they're useful in programming.

You will then move on to describe how you would pass information to and get information from functions. And finally, try and list out the differences between positional arguments and keyword arguments in functions.

It's a good time for you to now pause this video for a bit. Think about these questions and maybe jot down a few answers before we move on to discussing sample solutions.

[Video description begins] *Solution* [Video description ends]

The simplest definition of a function that is the most intuitive is the fact that it is a piece of a reusable code. It is code that you write once which you can use over and over again. Not just you, once you define a function, other developers can use that code as well.

This bit of reusable code can be used for anything. It can perform an action or a calculation, such as adding two numbers, subtracting a number from the other. This bit of reusable code can be customized to work on different inputs.

You can pass in different inputs to a function and have your function act on those inputs. And once your function has performed the calculation or operation, it's possible for the function to return a result to the user.

In this way, you're kind of modularizing your code where you break it up into little pieces and have functions work on these little pieces. In programming, creating functions with reusable bits of code is an important aspect of writing maintainable and modular code.

Let's move on. Defining a function in Python is just definition. It does not actually invoke or execute the code that lies within the function. A function definition essentially says, here is a function with this name that performs these actions.

But the actions are not executed till you actually go ahead and invoke the function. Once you've defined the function, it can be invoked whenever you want that action or calculation performed. And the code and the function body, that is the code associated with the function, will be executed when you invoke the function.

Now, your answers, of course, may not be exactly the same. This is just a sample solution, but as long as they cover the basic concepts, you should be good. We'll now move on to discussing how you can pass information to a function.

When you define your function, you can specify that it accepts input arguments. Input arguments are variables which you can use to pass information to a function. The function can then perform its operation or calculation based on the information that you pass in.

At the time, when you define the function. That is when you specify what code should be executed in the function body. That's when you define the input arguments that you pass into a function as well. Along with the def keyword, the name of the function.

That's where you specify the input arguments. Just like the ordinary variables that we worked with so far in Python, input arguments are just variables. And these can be accessed by the function to perform its action or calculation.

A function can act on its input arguments. When a function is invoked. The user or the invoker of the function specifies values for the input arguments that are associated with the variables of the input arguments. What kind of information can you pass to a function?

Well, anything, input arguments can be of any data type. Primitive types such as floats or integers. Complex types such as lists, dictionaries, and even custom types. You can pass information to a function. You can also receive information or results from a function in the form of return values. Information is returned from a function using the return statement.

Now it's possible that a function does not have a return statement. In that case, it's implicitly assumed that the default return value from this function is the special Python value, None. When you have your function perform an operation or a calculation, there is probably a result.

And this is a result that you might need in other parts of your program. This is the result that your function might return. Just like input arguments, the return values from functions can be any kind of data.

Primitive types, such as strings, floats, integers or Boolean values. Complex types, such as lists, sets, dictionaries, or even your own custom data type. We'll now move on to discussing characteristics of positional arguments and keyword arguments.

Positional arguments are input arguments to a function that are specified along with the function definition. Positional arguments are so-called because values are assigned to these arguments based on their position in the function definition.

The first value is assigned to the first input argument, the second value to the second input argument, and so on. Values are assigned at the time of function invocation. Function invocation is when values are associated with specific arguments.

Positional arguments differ from keyword arguments in that the arguments are not referred to by variable names. The order of the values should match the order of arguments in the case of positional arguments, unless default values are specified. Positional arguments are required arguments.

And positional arguments should always be specified before keyword arguments. We'll now move on to discussing keyword arguments. There is no change in how you define your custom function, whether you use positional arguments for invocation or keyword arguments.

Values are assigned to keyword arguments by variable names at the time of function invocation. The most important difference between position and keyword arguments. Is the fact that keyword arguments are referred to by the name of the variable.

The order of the values in keyword arguments need not match the original order of the arguments in the function definition. Keyword arguments can be specified in any order since they're associated using the variable name.

Unless default values are specified, keyword arguments are required arguments, just like positional arguments. And keyword arguments should always be specified after positional arguments.