Functions in Python: Working with Advanced Features of Python Functions

This course explores advanced Python function topics such as recursion, closures, and using generator functions to generate sequences. In 12 videos, you will learn how to use decorators to add functionality to code; examine how recursion can be used to construct code to solve complex problems; and learn to write a terminating condition for a recursive function. Next, you will learn how to use an Iterator to respond to a built-in next () function. Learners will also examine closures, and how as functions they maintain their own lexical environment; and explore how closures are functions that can yield dramatic results in the distributed processing of code, and are widely used in the implementation of distributed processing frameworks. Then you will learn how to use generator functions to generate sequences. You will learn how sequences can iterated upon by other parts of your program. Finally, you will learn that using decorators offers simple ways of invoking higher-order functions.

Table of Contents

Course Overview

[Video description begins] *Topic title: Course Overview. Your host for this session is Janani Ravi, a software engineer and big data expert*. [Video description ends]

Hello there, welcome to this course. My name is Janani Ravi and I will be your instructor for this course. A little about myself first. I am the co-founder of Loonycorn, a studio for high quality video content. Before founding Loonycorn, 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 paralyzed and shipped off for distributed execution relatively easily. In this course, working with advanced features of Python functions, you will cover a host of advanced topics related to functions in Python. You will see how recursion can be used to construct extremely elegant code to solve complex problems. You will learn about closures, which are functions that effectively carry around their own state within them. You will learn how closures are effectively functions that maintain their own lexical environment. Closures can yield dramatic results in the distributed processing of code, and are widely used in the implementation of distributed processing frameworks.

You will then move on to the use of generator functions, which are a great way to generate sequences. Those sequences can then be iterated upon by other parts of your program. Finally, you will learn the use of decorators, which offers simple ways of invoking higher order functions. By the end of this course, you will be able to identify situations that call for the use of advanced features of functions in Python, such as closures, generators and decorators. And use these features as well as functional recursion to correctly implement functions for your use case.

Recursion - Invoking Functions

[Video description begins] *Topic title: Recursion - Invoking Functions. The presenter is Janani Ravi.* [Video description ends]

We've studied in earlier videos that functions can invoke other functions to get their job done.

[Video description begins] *She opens the Recursion file in the Jupyter Notebook window. She opens the first diagram: def hello(name) function is pointing to the print("Hello", name) command, which is connected to hello(name). The hello(name) is further pointing to def hello(name). The Hello Ron output is displayed adjacent to print("Hello", name).* [Video description ends]

We saw how the calculate function can invoke other functions such as add, subtract, multiply, and divide to perform a specific arithmetic operation.

[Video description begins] *The second diagram is displayed: In the first box, the following text is displayed: num = 1 num + 1. In the second box, the following text is displayed: num = 2 num + 1. In the third box, the following text is displayed: num = 3 num + 1. In the fourth box, the following text is displayed: num = 4 num + 1. In the fifth box, the following text is displayed: Infinite Terminated. An arrow is displayed from the first box pointing to the second box. Besides this arrow, number 2 is displayed. An arrow is displayed from the second box pointing to the third box. Besides this arrow, number 3 is displayed. An arrow is displayed from the third box pointing to the fourth box. Besides this arrow, number 4 is displayed. An arrow is displayed from the fourth box pointing to the fifth box. Besides this arrow, number 5 is displayed.* [Video description ends]

If functions can invoke other functions it stands to reason that a function can invoke itself. And that's exactly what Recursion is all about.

[Video description begins] *The following message is displayed: Recursion Invoking a function from within the function itself.* [Video description ends]

The idea behind Recursion is simple, but it's a rather mind-bending concept. And once you get down to the implementation, you might find that it's a little hard to understand. Well, the only thing is practice and try out problems on your own. That's what will get you better with Recursion. Let's first start with a simple example, where we have a function which invokes itself from within the body of the function.

[Video description begins] *She enters a code in the first code cell. The code is, code starts: def hello(name): print("Hello", name) hello(name). Code ends.* [Video description ends]

Here, I have a function named hello which takes in one input argument that is name. Within this function, I simply print out hello and the name. Now, before I return from the function, I say, let me go ahead and invoke hello once again. Now after having defined this function when I execute it, what do you think will happen? Think for a bit, the call to the hello function from within the hello function body is called a recursive call. And observe it's just like any other function invocation of hello, we pass in name. I'm going to go ahead and invoke the hello function that we just defined.

[Video description begins] *She enters a code in the second code cell. The code is: hello("Ron").* [Video description ends]

Hit Shift+Enter, and boom, you're headed into an infinite loop which goes on, and on, and on, till you encounter an error.

[Video description begins] *She executes the code in the second code cell. The output displays the text: Hello Ron infinite number of times. At the end, the following error message is displayed: RecursionError: maximum recursion depth exceeded while calling a Python object.* [Video description ends]

What exactly is this error, and what exactly happened here? The error is specifically a RecursionError, the maximum Recursion depth was exceeded while calling a Python object. So, why did we encounter this error, and what exactly happened? The first time you invoked hello and passed in Ron, Ron was printed to screen.

[Video description begins] *The first diagram is displayed.* [Video description ends]

Then before you could end the function, you called hello once again. This caused Ron to be printed to screen once again. Then once again, you called hello before you ended the function and this went on. You might have encountered infinite loops when you worked with while loops earlier. This is an example of an infinite function invocation. You're recursing an infinite number of times.

[Video description begins] *The following message is displayed: Infinite recursion Can cause you to run out of memory and crash.* [Video description ends]

It so happens that this can cause you to run out of Python memory because for every invocation, Python has to allocate some memory for function execution. And in order to prevent this kind of crash and out of memory problems, Python has a built in limit on the number of times you can recursively invoke a function. Once this limit is reached, this recursion error is what you encounter. So let's see what this limit is and how you can configure it.

[Video description begins] *She enters a code in the third code cell. The code is: import sys sys.getrecursionlimit().* [Video description ends]

Import the sys module which gives you system specific information, that is information about your Python running on your machine and call sys.getrecursionlimit(). You can see that the default recursion limit is 3000.

[Video description begins] *She executes the code in the third code cell. The output reads: 3000.* [Video description ends]

That means you can invoke a function recursively, that a function can call itself 3000 times before it encounters an error. Let's see this recursive limit in action once again, but this time with a different function.

[Video description begins] *She enters a code in the fourth code cell. The code is, code starts: def increment(num): print(num, end = " ") increment(num + 1) return. Code ends.* [Video description ends]

This function is named increment, and it takes in one input argument, a number. The first thing that we do is print the number out to screen. And instead of using a new line, which is the default after a print, we terminate the print using the space character. The numbers that we print to screen will be printed one after the other in a row. We then invoke the increment function once again. This is a recursive call, but this time, we've changed the value of the input argument that we pass into increment.

If the original number that was passed in was 1, this time we'll pass in 1 + 1, that is 2. Each time we invoke the increment function, we increment the number that we pass in. So you will see that the number printed out to screen will start with whatever number you specified first and each display will be incremented by one. Once we have defined this increment recursive function, I'm going to go ahead and invoke it.

[Video description begins] *She enters a code in the fifth code cell. The code is: increment(1).* [Video description ends]

And the first value of num that I pass in is equal to 1. I'm going to hit Shift+Enter, execute this function. And let's analyze what happens after we see the result. Observe that the numbers 1, 2, 3, 4 in sequential order is printed out to screen. And this sequential printing of numbers goes on all the way up to 2954, so almost 3000. And once we get to around this range, you can see that we encounter the same RecursiveError, maximum Recursion depth exceeded.

If your Recursion depth is the same, that is, 3000, and you run this code, you might find that the final number printed out will be a little different. It also depends on your input buffering and so on. So there is some random element involved there. But you'll see that the last number will be somewhere close to 3000. In order to wrap our heads around a Recursion, let's understand what exactly happened when I made this recursive call.

[Video description begins] *The code of the fourth code cell is displayed*. [Video description ends]

In the first invocation of the increment function, the value of the num input argument was equal to 1 and we printed out 1 to screen.

[Video description begins] *The following message is displayed: num = 1 increment(1+1)*. [Video description ends]

Once num was printed out, we invoked increment once again, but this time we updated the input argument and we invoked increment by passing in 1 + 1 which is equal to 2. So the next time we enter this function, the value of the num input argument is equal to 2, and we print out 2 to screen.

[Video description begins] *The following message is displayed: num = 2 increment(2+1)*. [Video description ends]

We invoke increment once again, but we pass in 2 + 1. So the third time this function is invoked, the value of num that is passed in is 3. 3 is printed out to screen and we invoke the increment function once again, and this time we pass in 3 + 1.

[Video description begins] *The following message is displayed: num = 3 increment(3+1)*. [Video description ends]

The next time increment is invoked, the value of num is 4. And this will go on and on to infinity in exactly the same way. Each time the value of num is incremented, observe that we never return from our original function invocation when the value of num was equal to 1.

[Video description begins] *The second diagram is displayed*. [Video description ends]

That's because we are getting deeper and deeper into the same function. And this continues till the recursive limit is reached and Python forcefully terminates the execution of this function.

[Video description begins] *She enters a code in the sixth code cell. The code is: sys.setrecursionlimit(100)*. [Video description ends]

Now, if you have a particular bit of Python code for which you need a different recursion limit, you can change it by invoking sys.setrecursionlimit. Here, I've set my recursion limit to 100. I'm going to invoke the increment function once again, starting off at 1 with this updated recursion limit.

[Video description begins] *She enters a code in the seventh code cell. The code is: increment(1)*. [Video description ends]

Hit Shift+Enter and you'll find that once again, you encounter a RecursionError. Your function is invoked roughly 100 times and then Python stops execution. The last number that you see printed out within your increment function may not be close to a 100, may not be exactly 100.

[Video description begins] *She executes the code in the seventh code cell. The output displays numbers from 1 to 54 along with the RecursionError message.* [Video description ends]

This is because there is some buffering going on at the output and it's possible that Python just didn't print the last few numbers before the error was encountered.

Recursion - Conditions

[Video description begins] *Topic title: Recursion - Conditions. The presenter is Janani Ravi.* [Video description ends]

What's the use of a recursive function call if it's going to cause an error?

[Video description begins] *She opens the Recursion file in the Jupyter Notebook window.* [Video description ends]

Well, the way to avoid this is to have some kind of terminating condition. While writing a recursive function, it's extremely important that you specify a terminating condition when you'll no longer recursively call your function.

[Video description begins] *The following message is displayed: Terminating condition Ensure that your recurve invocation is terminated based on some condition.* [Video description ends]

You'll simply stop and return, this is the first thing you have to think of when you write your recursive function. Before we move on to writing recursive functions with a terminating condition, let's take a look at a simple non-recursive function. The decrement function takes as an input argument, a number, prints the number to screen, decrements the number, and prints a decrementing sequence all the way up to 1.

[Video description begins] *She enters a code in the eighth code cell. The code is, code starts: def decrement(num): while num > 0: print(num) num = num - 1. Code ends.* [Video description ends]

As soon as the number is 0, it'll stop. Observe that this function uses a while loop. While loops also have terminating conditions. Observe that the condition that we have specified for while loop is that it will iterate, so long as the value of num is greater than 0. We decrement num every iteration of this while loop. As soon as num reaches 0, the while loop will terminate. That is the terminating condition of this while loop. I'm going to go ahead and invoke this decrement function and pass in 10.

[Video description begins] *She enters a code in the ninth code cell. The code is: decrement(10).* [Video description ends]

And print it out to screen I get 10, 9, 8 all the way through to 1. As soon as the variable num got to the value 0, the while loop was terminated and nothing was printed to screen.

[Video description begins] *She executes the code in the ninth code cell. The output reads: 10 9 8 7 6 5 4 3 2 1.* [Video description ends]

Let's now right the same decrement function, but recursively. We won't use a while loop, we'll simply invoke the same function over and over again. And each time we'll decrement the number that we pass in by 1. Let's take a look at the code for this decrement recursive function. We have the decrement function, it takes as an input argument a number, no different from earlier functions that we've seen. The first thing to do is to think of the terminating condition for this recursive function.

[Video description begins] *She enters a code in the tenth code cell. The code is, code starts: def decrement(num): if num == 0: return print(num) decrement(num - 1). Code ends.* [Video description ends]

Before you write any other code, the terminating condition is the most important, otherwise we'll get infinite recursion. The terminating condition here is, if the number that is passed in is equal to 0, we simply return. We don't print anything to screen, we don't invoke the decrement once again. So we terminate and get out of the function call once num is equal to 0. The rest of the code in this function body will be executed only when num is a value greater than 0.

So if num is greater than 0, we'll print num out to screen and then invoke the decrement function recursively. But this time we'll pass in num - 1 as an input argument. So if you call num with a value of 5, 5 is not equal to 0, so we go on. 5 will be printed to screen, then decrement will be called again with 5 - 1, which is 4. When decrement is called with 4, 4 will be printed to screen, we'll call decrement with 3 and so on.

[Video description begins] *A diagram is displayed: In the first box, the following text is displayed: num = 5 num - 1. In the second box, the following text is displayed: num = 4 num - 1. In the third box, the following text is displayed: num = 3 num - 1. In the fourth box, the following text is displayed: num = 2 num - 1. In the fifth box, the following text is displayed: num = 1 num - 1. In the sixth box, the following text is displayed: num == 0 Terminated*. [Video description ends]

We'll now go ahead and invoke that decrement function and pass in 10 as the initial value for num. Observe that 10, 9, 8, 7, 6, 5 is printed to screen.

[Video description begins] *She enters a code in the eleventh code cell. The code is: decrement(10). She executes the code in the eleventh code cell. The output reads: 10 9 8 7 6 5 4 3 2 1*. [Video description ends]

As soon as 0 is passed into the decrement function, we return without invoking decrement once again, that is the terminating condition, the recursion ends. Now you understand how the decrement function is invoked repeatedly.

[Video description begins] *A diagram is displayed: In the first box, the following text is displayed: num = 10 num - 1. In the second box, the following text is displayed: num = 9 num - 1. In the third box, the following text is displayed: num = 8 num - 1. Besides the fourth box, the following text is displayed: Up to num = 1. In the fifth box, the following text is displayed: num = 1 num - 1. In the sixth box, the following text is displayed: num == 0 Terminated*. [Video description ends]

And how we go from printing 10, then 9, then 8, because each time we invoke decrement, we decrement the value we pass in as an input argument. Let's now discuss how the terminating condition works. At some point, the value of num that we pass in will be equal to 0.

[Video description begins] *The following message is displayed: num = 0 If condition will evaluate to true and the function return*. [Video description ends]

At that point, if num is equal to 0, that condition will be satisfied. Each check that we have performed at the beginning of the decrement function will evaluate to true and we will return from decrement without invoking decrement further. Once we return from this invocation of decrement, we've ended the previous invocations of decrement as well. So you can think of it as a rope unrolling, all of the invocations are now complete and the function terminates. Once you get more comfortable with programming, you'll see that recursion is actually an extremely cool concept and is widely used in the real world because it allows you to break larger problems into smaller ones and write clean, elegant code. Here is another example of a recursive function called a recursive_sum.

[Video description begins] *She enters a code in the twelfth code cell. The code is, code starts: def recursive_sum(num): if num == 0: return 0 result = num + recursive_sum(num - 1) return result. Code ends*. [Video description ends]

Whatever number you pass in, it'll calculate the sum of all numbers all the way through that number. So if you pass in 5, it'll calculate 1 + 2 + 3 + 4 + 5. Let's examine this recursive function and see how this is done.

[Video description begins] *The following message is displayed: recursive_sum(5) 5+4+3+2+1*. [Video description ends]

When you think about writing a recursive function, the first thing to consider is the terminating condition. When do you want this function to stop calling itself over and over again? Well, in my case, once again, the terminating condition is when num that is passed in is equal to 0. I only want to sum up all of the positive integers, so if num is equal to 0, I'll return 0. Another change that I made to this recursive function is the fact that it has a return value. So if num is equal to 0, we won't invoke the function once again, we'll simply return 0. The rest of the code in the function body will be executed only when the input argument num is greater than 0, and what's that code? The first thing we do is we calculate num + recursive_sum (num - 1).

[Video description begins] *The following message is displayed: result = 5 + recursive_sum(4) recursive_sum(4) = 4+3+2+1*. [Video description ends]

So if the original number num was 5, the result will be 5 + recursive _sum(4). Now, we know that the recursive_sum(4) is 4 + 3 + 2 + 1. So 5 + recursive _sum(4) will give us the result that we need. Observe that we have made a recursive call to the recursive_sum function with the input argument 4.

[Video description begins] *The following message is displayed: num = 4 result = 4 + recursive_sum(3)*. [Video description ends]

So the next time we enter this recursive_sum function, num is equal to 4, num is not 0, so the terminating condition is not satisfied. And result is equal to 4 + the recursive_sum of 3. We invoke the recursive_sum function with an input argument of 3.

[Video description begins] *The following message is displayed: num = 3 result = 3 + recursive_sum(2)*. [Video description ends]

So we enter this function once again, num = 3. And the result that is calculated is 3 + recursive_sum(2). Observe how we are breaking down the problem, the recursive_sum is now called with num is equal to 2.

[Video description begins] *The following message is displayed: num = 2 result = 2 + recursive_sum(1)*. [Video description ends]

So the result is 2 + recursive_sum(1). We enter the recursive_sum function once again, this time the value of num equal to 1, the terminating condition is yet not satisfied.

[Video description begins] *The following message is displayed: num = 1 result = 1 + recursive_sum(0)*. [Video description ends]

The result is 1 + recursive _sum(0). Now finally, when we invoke the recursive_sum function with num = 0, it simply returns 0.

[Video description begins] *The following message is displayed: num = 0 return 0*. [Video description ends]

We don't go deeper into the invocation of the function. Now we come to the unrolling phase of the function invocation, let's start unrolling the function.

[Video description begins] *The following message is displayed: num = 1 result = 1 + recursive_sum(0) = 1 + 0 = 1*. [Video description ends]

Remember, the last invocation we made to recursive _sum, where we passed in num = 0 was when we were trying to calculate result of 1 + recursive_sum(0). We know recursive_sum (0) is 0, so we get the result here equal to 1. This function invocation, when the input argument num was equal to one, will return the value of 1 to whoever called that function. And that was the recursive_sum function itself when num was equal to 2.

[Video description begins] *The following message is displayed: num = 2 result = 2 + recursive_sum(1) = 2 + 1 = 3*. [Video description ends]

We know the answer to recursive_sum(1) now, which is just 1, so the result here is 2 + recursive_sum(1), which is 2 + 1 = 3. This result of 3 is returned to the caller of this version of the function. And the caller of this function is the recursive_sum function itself, when the input argument num was equal to 3.

[Video description begins] *The following message is displayed: num = 3 result = 3 + recursive_sum(2) = 3 + 3 = 6*. [Video description ends]

In our unrolling, we have the value of recursive_sum(2) at this point, we know that that result is 3. So here we calculate 3 + 3, the result is 6. And this value of 6 is returned to the caller of this version of the function.

[Video description begins] *The following message is displayed: num = 4 result = 4 + recursive_sum(3) = 4 + 6 = 10*. [Video description ends]

And that was the recursive_sum function when the input argument num was equal to 4. So the result of 4 + recursive_sum(3) is what we were trying to calculate. We know that the recursive_sum(3) is equal to 6, so we calculate 4 + 6 here, we get the result 10. And this result 10 is returned to the invoker of this version of the function. And now, our unwinding and unrolling is almost complete.

[Video description begins] *The following message is displayed: num = 5 result = 5 + recursive_sum(4) = 5 + 10 = 15*. [Video description ends]

The version of the function we are at is our original invocation of recursive_sum with num = 5. And the result is 5 + recursive_sum(4). We know that the recursive_sum(4) is 10 which gives us 5 + 10, 15, and that is the final result. This final result is what is returned to us in the main program outside the function. Now, let's try out recursive_sum with 3 and we get the result 6 as expected.

[Video description begins] *She enters a code in the thirteenth code cell. The code is: recursive_sum(3). She executes the code in the thirteenth code cell. The output reads: 6*. [Video description ends]

Let's try out recursive_sum once again, with a different input argument, let me try 10.

[Video description begins] *She enters a code in the fourteenth code cell. The code is: recursive_sum(10). She executes the code in the fourteenth code cell. The output reads: 55*. [Video description ends]

This will give me 10 + 9 + 8, all the way through 1 and the answer is 55. When you're first starting with programming, recursion is hard. I suggest you work these out on paper to see how we get the final result. Figure out the invocation calls and the unrolling of the invocations.

Recursion - Calls

[Video description begins] *Topic title: Recursion - Calls. The presenter is Janani Ravi*. [Video description ends]

One of the most common problems that is solved in a recursive manner is to find the factorial of a number. You'll see that the code to find the factorial of a number is very straightforward. Once you understand how terminating conditions and return values work in recursion. We've already covered that in a lot of detail in the previous video. Which is why we are now ready to tackle the code which calculates the factorial of a number recursively.

[Video description begins] *She opens the Recursion file in the Jupyter Notebook window*. [Video description ends]

Here is the factorial function that I have defined, it takes as an input argument a single number. Remember, whenever you're writing a function that is going to call itself that is a recursive function. The first thing that you

ought to put in is the terminating condition. Factorials can be calculated only for positive numbers. Which is why I'm going to put an error check first, I'm going to check whether the number that's passed in is positive or not. If it's less than 0, that is it's negative, I'm going to print, factorial does not exist for negative numbers, and simply return. This is not a terminating condition, rather, it's an error check. The next if-statement here within this function body is our terminating condition.

[Video description begins] *She enters a code in the fifteenth code cell. The code is, code starts: def factorial(number): if number < 0: print("Sorry, factorial does not exist for negative numbers") return if number == 0: return 1 return number * factorial(number - 1). Code ends.* [Video description ends]

If the number that is passed in to this invocation of factorial is equal to 0, we know that the factorial of 0 is 1. And we simply return 1 to the invoker of this function. Remember, the whole idea of the terminating condition is that you don't invoke this function anymore times. You simply return with some return value if needed. Because you're using recursion to break more complex problems into simple one, the actual body of the factorial code is very straightforward. We simply multiply the current number that is passed in by the factorial of the number - 1. The reason why this very simple and elegant code works and why it helps you find the factorial of a number. Is because of the recursive nature of the factorial calculation.

[Video description begins] *The following message is displayed: factorial(3) = 3*2*1 factorial(3) = 3 * factorial(2).* [Video description ends]

We know that the factorial of 3 is basically 3*2*1, this is the same as 3 * factorial(2). Remember, the factorial of 2 is 2*1. Let's walk through this factorial code using this simple example. Let's say we invoke this factorial function and pass in 3 as the number.

[Video description begins] *The following message is displayed: number = 3 factorial(3) = 3 * factorial(2).* [Video description ends]

So for this invocation of factorial, num is equal to 3, you want to find factorial of 3, we essentially say it's 3 multiplied by the factorial of 2. So we invoke the factorial function once again, and number is 2 this time around. In the recursive invocation of the factorial function, the number input variable is equal to 2.

[Video description begins] *The following message is displayed: number = 2 factorial(2) = 2 * factorial(1).* [Video description ends]

We now want to calculate the factorial of 2. The error condition is not met, the terminating condition is not met. So essentially we'll get to the last line of this function where we see the factorial of 2 is 2 * factorial(1). We now enter another recursive call to the factorial function where the input argument number = 1.

[Video description begins] *The following message is displayed: number = 1 factorial(1) = 1 * factorial(0).* [Video description ends]

Number is not less than 0, so the error condition is not satisfied. Number is not equal to 0, the terminating condition is not satisfied. So we get to the last line of this function where we say that the factorial of 1 is 1 multiplied by the factorial of 0. And this brings us to the next recursive invocation of the factorial function, this time the value of the input argument number is equal to 0.

[Video description begins] *The following message is displayed: number = 0 return 1.* [Video description ends]

If you observe the code of this function, this is our terminating condition, if number is equal to 0, we return 1. This return statement basically means that we are in the unrolling or unlooping portion of our recursive invocations. We return 1 from this invocation of factorial, which means we go back to the previous invocation.

[Video description begins] *The following message is displayed: number = 1 factorial(1) = 1 * factorial(0) = 1 * 1 = 1.* [Video description ends]

The previous invocation was our invocation of factorial with number equal to 1. Factorial of 1 we said was equal to 1 multiplied by factorial of 0, we know that factorial of 0 is equal to 1. So we get 1 multiplied by 1 equal to 1, and this is what is returned from this particular invocation of the factorial function. So we go now to the previous invocation. The previous invocation was where the input argument number was equal to 2, calculating the factorial of 2 is 2 multiplied by the factorial of 1.

[Video description begins] *The following message is displayed: number = 2 factorial(2) = 2 * factorial(1) = 2 * 1 = 2*. [Video description ends]

We already have the factorial of 1 at this point, so we have 2 multiplied by 1 equal to 2. This is the factorial of 2, and this is what is returned to the previous invocation. As we unroll to the previous invocation of factorial, this was when the input argument number was equal to 3.

[Video description begins] *The following message is displayed: number = 3 factorial(3) = 3 * factorial(2) = 3 * 2 =6*. [Video description ends]

This is our original invocation. We wanted to find the factorial of 3, we already have the factorial of 2 at this point in time. Factorial of 2 we know is 2, so the factorial of 3 is 3 multiplied by 2. The final result is 6, and this is what is returned as the final solution. Now that we understood how factorial works, let's invoke this function to find the factorial of 3.

[Video description begins] *She enters a code in the sixteenth code cell. The code is: factorial(3). She executes the code in the sixteenth code cell. The output reads: 6*. [Video description ends]

The result we get is 6, and you can try this with different values as well.

[Video description begins] *She enters a code in the seventeenth code cell. The code is: factorial(5). She executes the code in the seventeenth code cell. The output reads: 120*. [Video description ends]

And for your understanding, you might want to work this out on your own to see how this is calculated. Factorial of 5 as we know is 120, 120 is returned from this function and printed to screen.

[Video description begins] *She enters a code in the eighteenth code cell. The code is: factorial(10). She executes the code in the eighteenth code cell. The output reads: 3628800*. [Video description ends]

Let's calculate the factorial of 10 and this tells us that it's 3628800. Let's do one last problem using recursion before we move on to other topics.

[Video description begins] *She enters a code in the nineteenth code cell. The code is: code starts: def fibonacci(number, fib_series): if number < 2: return l = len(fib_series) new_number = fib_series[l – 1] + fib_series[l – 2] fib_series.append(new_number) fibonacci(number – 1, fib_series). Code ends*. [Video description ends]

This once again, is a famous example of a recursive function calculating the Fibonacci series. The Fibonacci series is a very famous series in mathematics, this is what it looks like.

[Video description begins] *The following message is displayed: 1, 1, 2, 3, 5, 8, 13, 21 ….. Each number is the sum of the previous 2 numbers*. [Video description ends]

Now see if you can see a pattern within these numbers, observe that every number is the sum of the previous two numbers. Now because of the way the Fibonacci series is created, it lends itself beautifully to recursion. This function here will use recursion to calculate and display the Fibonacci series. I'm not going to walk you through every function invocation. I'll talk about how this code is structured. And I'll leave it to you as a self-exercise to walk through every invocation to see exactly what happens when you use this code. The name of this function is fibonacci and it takes in two input arguments.

The first argument, number, refers to the number of elements that you want to generate in the Fibonacci series. The second argument, fib_series, is a Python list. This fib_series Python list will be initialized with the first two numbers in the Fibonacci series. We have to start somewhere, and additional elements as we calculate them will be appended to this Python list. This fib_series list will contain the final Fibonacci series. In any recursive function, the first thing you need to specify is the terminating condition. If number is less than 2, we'll simply return. We are going to see the Fibonacci series with the first two numbers. Which is why if the number of elements we want is less than 2, we can just return.

We already have two elements. Now in order to calculate the next element in the Fibonacci series, we need to access the last two elements in the series. l here I'm going to initialize to the current length of the Fibonacci series, and this is obtained by calculating the length of the fib_series list. The new number in the fibonacci series stored in the variable new_number is the sum of the last two numbers. The number at l - 1 and the number at l - 2 index positions. We append this new number to the the fib_series list, and once we've generated this number, we simply invoke the Fibonacci function recursively.

But this time, we already have one of the numbers we want. So we invoke it with number - 1, this is what will allow us to hit our terminating condition. The recursive calls to the fibonacci function will constantly update the fib_series list. And the final fibonacci series that we're looking for will be available in this list. Let's invoke this function and see how it works.

[Video description begins] *She enters a code in the twentieth code cell. The code is: series = [0, 1] fibonacci(4, series).* [Video description ends]

I've initialize the series Python list with the first two elements in the series, 0, 1, and I invoke fibonacci. I want 4 numbers and I pass in the series Python list which is constantly updated. Once the recursive function has run through, let's print out what's in series now, and here is the fibonacci series all generated.

[Video description begins] *She enters a code in the twenty-first code cell. The code is: series. She executes the code in the twenty-first code cell. The output reads: [0, 1, 1, 2, 3].* [Video description ends]

Now if seemed confusing to you, let's try this once again, I'm looking for 10 fibonacci numbers.

[Video description begins] *She enters a code in the twenty-second code cell. The code is: series = [0, 1] fibonacci(10, series).* [Video description ends]

Now when I get the final series result, you'll see that all ten numbers are present here within this list.

[Video description begins] *She enters a code in the twenty-third code cell. The code is: series. She executes the code in the twenty-third code cell. The output reads: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55].* [Video description ends]

Remember, each recursive invocation to the fibonacci function updates this list, and that's why you have this final result here. If you want to debug your function and want to understand what's going on in the intermediate steps. You can redefine the function and add in a print statement.

[Video description begins] *She enters a code in the twenty-fourth code cell. The code is: code starts: def fibonacci(number, fib_series): if number < 2: return l = len(fib_series) new_number = fib_series[l - 1] + fib_series[l – 2] fib_series.append(new_number) print("Series so far", fib_series) fibonacci(number - 1, fib_series). Code ends. The code of the nineteenth code cell is displayed adjacent to the twenty-fourth code cell.* [Video description ends]

Observe that I have a print statement to have that says, Series so far. This gives me the fibonacci series for every invocation of this fibonacci function. Initialize the series python list to the first two elements of the series, 0,1, invoke fibonacci with 10.

[Video description begins] *She enters a code in the twenty-fifth code cell. The code is: series = [0, 1] fibonacci(10, series). She executes the code in the twenty-fifth code cell and the output is displayed.* [Video description ends]

And you'll see that at every step, the Fibonacci series is incremented by one new number. And each new number is the sum of the previous two numbers.

Generator Functions

[Video description begins] *Topic title: Generator Functions. The presenter is Janani Ravi.* [Video description ends]

Programming is all about having your program execute the logic that you have in mind. And a very important programming construct that facilitates implementation of such logic are control structures such as for and while loops. For and while loop allow you to iterate over collections of elements. But in fact, you can iterate over any sequence as long as that sequence has an iterator. What is an Iterator?

[Video description begins] *She opens the Generators file in the Jupyter Notebook window. The following message is displayed: Iterator Any object that responds to the built-in next() function.* [Video description ends]

It's a logical next question. An Iterator is any Python object that responds to the built-in next() function. There's a built in function called next, which when invoked on an object gives you the next element in a sequence. The sequence can be anything, next simply gives you the next element till you reach the end of the sequence. Essentially, an iterator is an object that allows you to access the next element in the sequence all the way from the beginning to the end of a sequence. And this is exactly what for loops and while loops need. Now that we know what an iterator is used for, we can talk about Generator functions.

[Video description begins] *The following message is displayed: Generator A function that is a simple way of creating iterators over a sequence.* [Video description ends]

A generator is essentially a function which offers a very simple way of creating iterators over any sequence. And generators in fact can be used to iterate over infinite sequences as well. The objective of a generator function is to create a sequence that can be iterated over. Now let's understand what exactly this means with an example, otherwise things are rather abstract.

[Video description begins] *She enters a code in the first code cell. The code is, code starts: def generator(): print('One!') yield 1 print('Two!') yield 2 print('Three!') yield 3 print('Four!') yield 4. Code ends.* [Video description ends]

A generator is a function and this function can have any name. I've called this function a generator but it can have any name like ABC, 1, 2, 3, anything. Before we understand what a generator does, let's understand the structure. Observe that this generator does not contain return statements, instead it contains the yield command. So one way in which a generator function is different from a regular function is the fact that it has the yield command within it. We have four yield commands here, yield one, two, three, and four. In order to understand this yield command, we need to understand what happens when a function is invoked in Python.

[Video description begins] *The following message is displayed: Invoking a function Control passes into the function till return is encountered.* [Video description ends]

There is an execution control in Python, and control of execution is then passed into the function when we invoke a function. The control remains within a function executing the code in the function body, till a return statement is encountered. And when a return is encountered, the execution control passes out of the function and back to the main program. The yield command is used to control this execution flow. And the execution flow

with yield is rather complicated, so it's best understood with an example. So observe the structure of this generator, we'll use it and I'll tell you exactly what's going on.

[Video description begins] *She enters a code in the second code cell. The code is: g = generator() g*. [Video description ends]

Now that we have our generator function defined, we can go ahead and invoke the generator and assign it to a variable. Generator functions differ from regular functions in that when you invoke a generator function, it creates a generator object.

[Video description begins] *She executes the code in the second code cell. The output reads: <generator object generator at 0x1095c9f48>*. [Video description ends]

Invoking the generator function give us this generator object stored in the G variable. However, the functions code has not been executed, unlike with regular functions. At this point in time even though the generator function was invoked, the code within it has not been executed. We only have the generator object. This generator object is actually an iterator which allows you to iterate over the sequence 1, 2, 3 and 4. Remember these are the return values of the yield commands. That is the sequence that we're going to iterate over. This generator object can be passed in as an input argument to the built-in next function, and this built-in next function allows us to iterate over our elements in sequence.

[Video description begins] *She enters a code in the third code cell. The code is: next(g)*. [Video description ends]

Let's talk about what happens when the next function is invoked on this generator. When you invoke Next, the generator function is actually executed all the way till the first yield statement, and the response from the generator is to print out 1 and return 1.

[Video description begins] *She executes the code in the third code cell. The output reads: One! 1*. [Video description ends]

Invoking next on this generator object executed all of the code till the first yield. print 1, yield 1, that's the only bit of code that has been executed so far. The whole idea of this generator is that you can access elements in a sequence.

[Video description begins] *She enters a code in the fourth code cell. The code is: next(g). The code of the first code cell is displayed adjacent to the fourth code cell*. [Video description ends]

Our sequence hasn't ended yet. There are additional yield statements in our generator function. The next time you invoke the next function on the g generator, it will remember that it has already executed the code till the first yield. It'll pick up the control flow of the execution starting after yield 1, it'll print Two, and yield 2.

[Video description begins] *She executes the code in the fourth code cell. The output reads: Two! 2*. [Video description ends]

2 is the return value from this function, and 2 is also the next element in the sequence. So for this invocation of Next, the only code that was executed was after yield 1 till yield 2. So we printed Two and returned 2. And 2 is the returned value, the next element in the sequence. Let's try this once again.

[Video description begins] *She enters a code in the fifth code cell. The code is: next(g)*. [Video description ends]

We invoke the next function and pass in the generator g. This time, the generator will remember that it has executed all of the code till yield 2. So it'll pick up the control flow of the execution starting after yield 2, it'll print out Three and yield 3. It'll return 3.

[Video description begins] *She executes the code in the fifth code cell. The output reads: Three! 3*. [Video description ends]

The use of this yield statement causes this generator function to remember where execution was stopped previously. Let's invoke the next function on this generator object once again.

[Video description begins] *She enters a code in the sixth code cell. The code is: next(g).* [Video description ends]

It'll pick up code execution after yield 3, Four will be printed to screen and we'll yield 4.

[Video description begins] *She executes the code in the sixth code cell. The output reads: Four! 4.* [Video description ends]

4 is what will be returned from the generator that is the next element in the sequence. Now that we've reached the last yield statement, we reach the end of our sequence. What if we try to invoke next on g now?

[Video description begins] *She enters a code in the seventh code cell. The code is: next(g).* [Video description ends]

Python goes into the generator function, sees that there are no more elements in the sequence, and it raises an error, the StopIteration error.

[Video description begins] *She executes the code in the seventh code cell. The output displays the following error message: StopIteration:.* [Video description ends]

If you want to iterate over the same sequence once again using this generator function, you have to invoke the generator once again, assign it to a variable.

[Video description begins] *She enters a code in the eighth code cell. The code is: g = generator() g. She executes the code in the eighth code cell. The output reads: <generator object generator at 0x108745840>.* [Video description ends]

Here is the variable g with the generator invoked afresh. Now you can invoke the next function on the generator and it will start off once again with One.

[Video description begins] *She enters a code in the ninth code cell. The code is: next(g). She executes the code in the ninth code cell. The output reads: One! 1.* [Video description ends]

One will be printed to screen, 1 was returned using the yield statement. If you continue invoking next on this generator object, you will go all the way through to Four, after which you will get the stop iteration error. What we just worked with was a very simple generator function. So I'm going to go one step further and redefine this generator function so that it's a little more complex. The main difference between this generator function and the previous one is the fact that it has a local variable equal to n.

[Video description begins] *She enters a code in the tenth code cell. The code is, code starts: def generator(): n = 1 print('One!') yield n n += 1 print('Two!') yield n n += 1 print('Three!') yield n n += 1 print('Four!') yield n. Code ends. The code of the first code cell is also displayed adjacent to the tenth code cell.* [Video description ends]

There are four yield statements within this generator function, indicating that there are four elements in the sequence that this generates. You can see that our entire function has four blocks which are separated by these yield statements. And each time the yield statement returns the value of n, the current value of n. I'm going to go ahead and invoke this generator function and assign the return generator object to the variable g, just like we did before.

[Video description begins] *She enters a code in the eleventh code cell. The code is: g = generator() g. She executes the code in the eleventh code cell. The output reads: <generator object generator at 0x108745c00>. The code of the tenth code cell is also displayed.* [Video description ends]

If you print out the value of g, you can see that it is a generator function object. The generator function object is an iterator that responds to the built-in next function. I'm going to invoke next on g. This is the first time that I've invoked next. What we get at the output is 1, and the return value is 1. How did this happen?

[Video description begins] *She enters a code in the twelfth code cell. The code is: next(g). She executes the code in the twelfth code cell. The output reads: One! 1.* [Video description ends]

Well, this was the first invocation of next, and Python remembers this. Python will execute all of the code within the generator function, up till and including the first yield statement. Python will initialize the local variable n to be equal to 1, print out 1 to screen, and yield n. The current value of n is 1, which is why this function first returns 1. I'm going to go ahead and invoke the next built-in function on this generator g. This is our second invocation. Python remembers that all of the code till the first yield statement has already been executed.

[Video description begins] *She enters a code in the thirteenth code cell. The code is: next(g). She executes the code in the thirteenth code cell. The output reads: Two! 2.* [Video description ends]

It also remembers this state, the last value assigned to the variable n. Python picks up the execution of this code after the first yield statement. So it'll increment n by 1, print Two, and return the current value of n which is 2. Let's invoke the next function on this generator object. Once again, Python remembers that all of the code up till the second yield statement has already been executed.

[Video description begins] *She enters a code in the fourteenth code cell. The code is: next(g). She executes the code in the fourteenth code cell. The output reads: Three! 3.* [Video description ends]

It'll then pick up execution from the statement after the second yield. It'll increment n by one, print Three to screen and return the current value of n which is 3. And finally, let's invoke next on this generator one last time. Python remembers that all of the code up until the third yield statement has already been executed.

[Video description begins] *She enters a code in the fifteenth code cell. The code is: next(g). She executes the code in the fifteenth code cell. The output reads: Four! 4.* [Video description ends]

It'll pick up execution from the code after the third yield statement which means it'll increment n by 1, print out Four to screen, and return the current value of n which is 4.

Generators for Infinite Sequences

[Video description begins] *Topic title: Generators for Infinite Sequences. The presenter is Janani Ravi.* [Video description ends]

Generators are extremely useful when you want to generate infinite sequences, when the entire sequence cannot fit in Python memory. So far we worked with very simple generators where we had explicit yield commands.

[Video description begins] *She opens the Generators file in the Jupyter Notebook window.* [Video description ends]

Generators are often used with for loops though or while loops. And that is exactly what we'll see here.

[Video description begins] *She enters a code in the sixteenth code cell. The code is, code starts: def generate_even_numbers(limit): for i in range(0, limit, 2): yield i. Code ends.* [Video description ends]

Generate_even_numbers is a generator function. How do you know it's a generator? Well, because of the yield statement that it has within the function body. Generate_even_numbers takes in one input argument, that is the limit. This function will generate even numbers starting from 0 up till that limit. Within the function body, we'll generate even numbers using a for loop. This is just much better than multiple yield statements. So for i in range(0, limit, 2) will give us all even numbers starting with zero up to, but not including limit and the step size

is 2. You already know how for loops work. Each new value is assigned to i, each value will be an even number and for every iteration of this for loop will yield i. Will return control from within this function to outside this function with the current value of i. Let's see this generator function in action.

[Video description begins] *She enters a code in the seventeenth code cell. The code is: g = generate_even_numbers(7).* [Video description ends]

I'm going to invoke generate_even_numbers and pass in 7. I want all even numbers up till 7. Remember that the function invocation for the generator does not actually execute the function. Only the generator object is created and assigned to the variable g. The object g is now an iterator and you can invoke the next function on g.

[Video description begins] *She enters a code in the eighteenth code cell. The code is: next(g).* [Video description ends]

When you invoke next what will happen is that the for loop will be initialized and it will start with 0.

[Video description begins] *She executes the code in the eighteenth code cell. The output reads: 0.* [Video description ends]

In the first iteration of the for loop, i will be equal to 0. When you call yield, i 0 will be returned to the caller. And this is why next(g) returns 0 in the first call.

[Video description begins] *She enters a code in the nineteenth code cell. The code is: next(g).* [Video description ends]

Now you can invoke the next function on g once again, and what will happen is that Python will remember that one iteration of this for loop is complete.

[Video description begins] *She executes the code in the nineteenth code cell. The output reads: 2.* [Video description ends]

So i is equal to 0 is complete. i should now be incremented to 2 and what you get here is yield i yield 2, and that is the return value. When you invoke next(g), once again Python will remember the last value assigned to i.

[Video description begins] *She enters a code in the twentieth code cell. The code is: next(g).* [Video description ends]

That is the state of the for loop. Where was it in its execution, i was 2, i is then incremented to 4, and yield i will return 4.

[Video description begins] *She executes the code in the twentieth code cell. The output reads: 4.* [Video description ends]

I'll invoke next(g) once again.

[Video description begins] *She enters a code in the twenty-first code cell. The code is: next(g).* [Video description ends]

This will run one more iteration of the for loop, and 6 will be returned to the caller.

[Video description begins] *She executes the code in the twenty-first code cell. The output reads: 6.* [Video description ends]

Now if I invoke next(g) once again, what will happen is that the for loop limit has been reached.

[Video description begins] *She enters a code in the twenty-second code cell. The code is: next(g). She executes the code in the twenty-second code cell. The output displays the following error message: StopIteration:. The code of the seventeenth code cell is displayed adjacent to the twenty-second code cell.* [Video description ends]

And this is why we get a StopIteration error. The previous invocation of the for loop was complete, there are no more elements in the sequence to yield. And that's why the StopIteration error was raised when you try to invoke next on this g generator object. If you want to iterate over even numbers once again, you need to invoke the generate_even_numbers function once again to get a new generator object, which I store in the g variable. So far, we've been manually invoking the next function to get elements in sequence.

[Video description begins] *She enters a code in the twenty-third code cell. The code is: g = generate_even_numbers(7) l = list(g).* [Video description ends]

But instead of that, you can simply pass in this generator object to the list function. And all of the elements in the sequence will be available to you in the form of a Python list.

[Video description begins] *She enters a code in the twenty-fourth code cell. The code is: l.* [Video description ends]

So print out l to screen and you can see that it contains all even numbers up to but not including 7.

[Video description begins] *She executes the code in the twenty-fourth code cell. The output reads: [0, 2, 4, 6].* [Video description ends]

When you pass in a generator object to the list function, the list function realizes that this is a generator, it invokes next until stop iteration is encountered. It collates all of the results together, creates a Python list and that is what is returned. Now that we understand how we can create a generator function that uses a for loop to generate the sequence, let's try another example. I'm going to use this function to generate squares of a number up till a certain limit.

[Video description begins] *She enters a code in the twenty-fifth code cell. The code is, code starts: def generate_squares(limit): for i in range(0, limit): yield i**2. Code ends.* [Video description ends]

In order to generate squares of numbers, we only need to change the specification of the for loop. We iterate from for i in range(0, limit). This will give us numbers in sequence starting from 0 and for each value of i, we yield i to the power 2. That is the square of i. The first step is to instantiate the generator object.

[Video description begins] *She enters a code in the twenty-sixth code cell. The code is: gen = generate_squares(10).* [Video description ends]

Invoke the generate squares function. I want squares of all numbers up to but not including 10. And store this generator object in the gen variable. Now that we have this generator object, we can invoke the next function on the generator to generate squares.

[Video description begins] *She enters a code in the twenty-seventh code cell. The code is: next(gen). She executes the code in the twenty-seventh code cell. The output reads: 0. The code of the twenty-fifth and twenty-sixth code cells is also displayed.* [Video description ends]

We start with 0, this is the first iteration of the for loop.

[Video description begins] *She enters a code in the twenty-eighth code cell. The code is: next(gen). She executes the code in the twenty-eighth code cell. The output reads: 1.* [Video description ends]

When we invoke next on this generator once again, I will have the value 1 and the square of 1 is 1.

[Video description begins] *She enters a code in the twenty-ninth code cell. The code is: next(gen). She executes the code in the twenty-ninth code cell. The output reads: 4.* [Video description ends]

Invoke next(gen) once again. I will now have the value 2. And the square of 2, as you can see here, is 4. That is what is yielded to us.

[Video description begins] *She enters a code in the thirtieth code cell. The code is: next(gen). She executes the code in the thirtieth code cell. The output reads: 9.* [Video description ends]

Next(gen) will give us 9. That is the square of 3. Instead of manually invoking next to access the elements in a sequence, you can use the generator object to generate a sequence, and then convert that sequence to a list.

[Video description begins] *She enters a code in the thirty-first code cell. The code is: squares_list = list(generate_squares(10)) squares_list.* [Video description ends]

So, generate squares will give us generator object, I pass the generator object to the list function. And the result will be a list of the square of numbers.

[Video description begins] *She executes the code in the thirty-first code cell. The output reads: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81].* [Video description ends]

If you want a tuple of the squares of numbers, all you have to do is to invoke the tuple function. Generate squares(10) will give us the generator object.

[Video description begins] *She enters a code in the thirty-second code cell. The code is: squares_tuple = tuple(generate_squares(10)) squares_tuple. Code ends.* [Video description ends]

That generator object we pass into the tuple function to give us all of the squares of the numbers up to but not including 10 in the form of a tuple.

[Video description begins] *She executes the code in the thirty-second code cell. The output reads: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81].* [Video description ends]

Now if you want an infinite loop you won't use the for controlled structure, instead you'll use a while loop to generate an infinite sequence.

[Video description begins] *She enters a code in the thirty-third code cell. The code is: code starts: def generate_powers_of_two(): num = 0 while True: num = num + 1 yield 2 ** num. Code ends.* [Video description ends]

And that's exactly what I've done here. Here is my generator function, it's called generate_powers_of_two. It takes a no input argument. Num is initially initialized to 0, and I have a while loop, which is just while True. So that means it runs through to infinity, it never ends. Typically, if your function had an infinite loop that would be a cause for concern. But if you want to generate an infinite sequence, well, you need an infinite loop. The code within this while True block increments the value of num by 1, and then yields 2 raised to the power num. So in the first iteration of this while loop, we'll get 2 to the power 1, then we'll get 2 to the power 2, 2 to the power 3 and so on. Let's see this infinite powers of two generator in action.

[Video description begins] *She enters a code in the thirty-fourth code cell. The code is: gen = generate_powers_of_two().* [Video description ends]

I'm going to first invoke the generate powers of 2 function. Remember this invocation will not execute the generator function. We'll get a generator object that we store in the gen variable.

[Video description begins] *She enters a code in the thirty-fifth code cell. The code is: next(gen). She executes the code in the thirty-fifth code cell. The output reads: 2.* [Video description ends]

When you first invoke the next function on this gen generator object, the first value of num is 1, 2 to the power 1 gives us 2. The yield statement within our generator function returns 2 to the power 1. Now if you invoke the next function on the generator object once again, we'll get into the next iteration of this while true loop.

[Video description begins] *She enters a code in the thirty-sixth code cell. The code is: next(gen). She executes the code in the thirty-sixth code cell. The output reads: 4.* [Video description ends]

Python remembers the previous value of num that existed, num was previously 1. We increment by 1 to get a value of num equal to 2, and we return 2 to the power 2, which is 4. There is one last cool thing you should know about a generator function.

[Video description begins] *She enters a code in the thirty-seventh code cell. The code is: count = 0 for p in gen: print(p) count = count + 1 if count > 10: break. Code ends.* [Video description ends]

A generator function can be iterated over using a for loop and that's exactly what we've done here. The gen variable holds the generator object that we had instantiated earlier.

[Video description begins] *The code of the thirty-third and thirty-fourth code cells is also displayed.* [Video description ends]

I've set up a for loop here for p in gen, to iterate over the entire sequence, then I print out p to screen. I don't want this to be an infinite sequence, which is why I have the count variable, which we count up to 10. As soon as count is greater than 10, I break out of this for loop.

[Video description begins] *She executes the code in the thirty-seventh code cell and the output is displayed.* [Video description ends]

Now if you execute this code, you will get all powers of 2 starting from where you left off. The last power of 2 we've got was 2 to the power 2 which is 4. Here, we start from 2 to the power 3, that is 8, 2 to the power 4, 16, 2 to the power 5, 32, and so on. This gen object starts at 8, that is, 2 to the power 3, because we'd already used the next function to generate the sequence, or 2 to the power 1, 2 and 2 to the power 2, 4.

Closures

[Video description begins] *Topic title: Closures. The presenter is Janani Ravi.* [Video description ends]

We've already seen that functions are first class citizens in Python. And generally, when a programming language creates a function just like other data types, that programming language supports something called Closures. And Closures are exactly what we're going to study here in this video.

[Video description begins] *She opens the Closures file in the Jupyter Notebook window.* [Video description ends]

The first thing you should know about a closure is the fact that it is a nested function. A nested function in that the definition for the closure is within an outer function.

[Video description begins] *She enters a code in the first code cell. The code is, code starts: def nested_hello_fn(): def hello(): print("Hello Cathy!") hello(). Code ends.* [Video description ends]

Let's take a look at this example here. Here, we have an outer function called nested_ hello_ function, it takes a no input argument. And within that, I have another function definition. This you know because I've used the def keyword here and defined another function called hello, and this nested function is defined within the outer function. This hello nested function only prints out, Hello Cathy to screen. I'm still within the outer function body. And after having defined the nested_hello_function, I invoke hello. The hello function that is defined here within the outer function is a closure.

[Video description begins] *The following message is displayed: Closures are nested functions Defined within an outer function*. [Video description ends]

Now, let's see how this works. I'm going to invoke the outer function first, nested_hello_fn() is its name.

[Video description begins] *She enters a code in the second code cell. The code is: nested_hello_fn()*. [Video description ends]

When I invoke the outer function, the hello function within it will be defined and then invoked because that's what is there in our function body. And you can see that Hello Cathy has been printed to screen.

[Video description begins] *She executes the code in the second code cell. The output reads: Hello Cathy!*. [Video description ends]

Hello has been defined within the outer nested hello function, which means if you try and invoke the hello function directly here from outside the function, that won't work.

[Video description begins] *She enters a code in the third code cell. The code is: hello(). She executes the code in the third code cell. The output displays the following error message: NameError: name 'hello' is not defined*. [Video description ends]

You can clearly see that there is a NameError here, hello is not defined. Well, if you want access to a function that is defined within another function, well, functions are first class citizens in Python, return the function object itself. Let's see how.

[Video description begins] *She enters a code in the fourth code cell. The code is, code starts: def get_hello_fn(): def hello(): print("Hello Cathy!") return hello. Code ends*. [Video description ends]

Here is an outer function called get_hello_function. Within this outer function, I've defined an inner function, or a closure, called hello. And this closure simply prints out Hello Cathy to screen. Instead of invoking this hello function, I simply return the hello function to whoever calls get_hello_function. So let's see this in action.

[Video description begins] *She enters a code in the fifth code cell. The code is, code starts: hello_fn = get _hello_fn() hello_fn. Code ends*. [Video description ends]

I invoke get_hello_function and store the return function object in the hello function variable. If you explore the contents of this hello function variable, you'll see that it is a function object.

[Video description begins] *She executes the code in the fifth code cell. The output reads: <function __main__.get_hello_fn.<locals>.hello()>* [Video description ends]

And observe the structure, you can see that it's the main module. It's within get hello function and it's a locally defined function, that is, a function defined within another function, that is a closure. Now, this closure can be invoked by using the hello_function variable.

[Video description begins] *She enters a code in the sixth code cell. The code is: hello_fn(). She executes the code in the sixth code cell. The output reads: Hello Cathy!*. [Video description ends]

Invoke hello function and that should print out Hello Cathy to screen. A closure is something beyond just an inner function defined within an outer function. There is more to it, we'll get to it in a step-by-step manner.

[Video description begins] *She enters a code in the seventh code cell. The code is, code starts: def greet_hello_by_name(name): def hello(): print("Hello!", name) hello() return hello. Code ends*. [Video description ends]

Here, I have an outer function called greet_hello_by_name, which takes in one input argument, the name of an individual. Within this outer function, I have an inner function, my closure. My inner function is called hello and it prints to screen Hello, and the name of the variable.

[Video description begins] *The following message is displayed: Closures can access local variables Local variables defined in the outer function*. [Video description ends]

Observe that this name variable is an input argument to the outer function. It's also accessible within the inner hello function. The name variable here can be thought of as a variable that is local to the outer function, local variables in the outer function can be accessed by closures. After having defined the closure, I invoke hello from within the function body of greet_hello_by_name. And I also return the hello function from the outer function. The last statement as you see here is a return hello. Now, let's go ahead and invoke this greet_hello_by_name function and the name that I've passed in is Chris.

[Video description begins] *She enters a code in the eighth code cell. The code is: greet_hello_fn = greet_hello_by_name("Chris")*. [Video description ends]

The function that is returned, I'm going to store in the greet_hello_fn() variable.

[Video description begins] *She executes the code in the eight code cell. The output reads: Hello! Chris*. [Video description ends]

Executing this prints out Hello Chris to screen. That's because we invoked the closure from within the outer function. We have a reference to the closure that was defined by the outer function.

[Video description begins] *She enters a code in the ninth code cell. The code is: greet_hello_fn. She executes the code in the ninth code cell. The output reads: <function __main__.greet_hello_by_name. <locals>.hello()>*. [Video description ends]

And this closure is available in the greet_hello_fn() variable. Let's see what happens if we invoke this greet_hello_fn(), and you can see that Hello Chris is printed out to screen.

[Video description begins] *She enters a code in the tenth code cell. The code is: greet_hello_fn. She executes the code in the tenth code cell. The output reads: Hello! Chris*. [Video description ends]

Notice something interesting here. Chris is available in the variable name which is local to the greet hello by name function.

[Video description begins] *The following message is displayed: Closures can access local state Even after the outer function has executed and exited*. [Video description ends]

Now, we've already invoked and exited greet_hello_by_name but the value in the name variable is still available to our closure. And this is another cool thing about closures in Python. They hold the reference to the local state even after the outer function that has define the local state no longer exist. Let's take a look at another slightly different example of this.

[Video description begins] *She enters a code in the eleventh code cell. The code is, code starts: def greet_by_name(name): greeting_msg = "Hi there!" def greeting(): print(greeting_msg, name) return greeting. Code ends*. [Video description ends]

This function, the outer function, is called greet_by_name and it takes in one input argument, name. Within the outer function, I have a local variable called greeting_msg which says, Hi there. I now have a closure called greeting defined within the outer function. And this greeting closure simply prints out the greeting_msg and the name to screen. Observe that it accesses the local variable the greeting_msg as well as the input argument name. And we return a reference to this greeting closure from the outer greet_by_name function.

[Video description begins] *She enters a code in the twelfth code cell. The code is: greet_fn = greet_by_name("Alex")*. [Video description ends]

Let's go ahead and invoke greet_by_name and store the function object that it returns in the greet_fn variable. We'll use this function object to greet Alex by name. Go ahead and invoke the greet_fn() by specifying parentheses. And it should say, Hi there, Alex. Observe how the closure has access not just to the name Alex but also to the greeting message, even after we have executed and exited the outer function.

[Video description begins] *She enters a code in the thirteenth code cell. The code is: greet_fn(). She executes the code in the thirteenth code cell. The output reads: Hi there! Alex*. [Video description ends]

Another way to express this is that closures carry around information about the local state. Let's do something a little drastic. I'm going to go ahead and delete the greet_by_name function, so you no longer have access to the outer function.

[Video description begins] *She enters a code in the fourteenth code cell. The code is: del greet_by_name*. [Video description ends]

Now remember that name and greeting message are both variables that were defined in the outer function. What happens to them? Well, we'll see in just a bit, go ahead and delete the greet_by_name function. Now if you try to invoke greet by name, Python should give you an error, greet_by_name is no longer defined.

[Video description begins] *She enters a code in the fifteenth code cell. The code is: greet_by_name('Alex'). She executes the code in the fifteenth code cell. The output displays the following error message: NameError: name 'greet_by_name' is not defined*. [Video description ends]

What about the greet_function?

[Video description begins] *She enters a code in the sixteenth code cell. The code is: greet_fn(). The code in the eleventh, twelfth, and thirteenth code cells is displayed*. [Video description ends]

Remember that the greet_function is a reference to our closure. Does this still work?

[Video description begins] *She executes the code in the sixteenth code cell. The output reads: Hi there! Alex*. [Video description ends]

And surprisingly enough, it does. Not only does it work, but it still has access to the local variables that were defined in the outer function. The outer function no longer exists in Python memory, but the local variables are still available along with our closure.

Closures and Local State

[Video description begins] *Topic title: Closures and Local State. The presenter is Janani Ravi*. [Video description ends]

Closures are very, very cool constructs, and they're rather mind-bending as well. So you need more practice to really grasp what's going on.

[Video description begins] *She opens the Closures file in the Jupyter Notebook window.* [Video description ends]

Let's take another example and work with closures. I've set up an import for the random module in Python.

[Video description begins] *She enters a code in the seventeenth code cell. The code is, code starts: import random def greet_with_personal_message(name, message): annotations = ['-', '*', '+', ':', '^'] annotate =*

*random.choice(annotations) def greeting(): print(annotate * 50) print(message, name) print(annotate * 50) return greeting. Code ends.* [Video description ends]

I'm going to use some functions within it to make some random choices. My outer function is called greet_with_personal_message, and it takes in two input arguments, name as well as message. In addition to the name and message, I'm going to decorate the personal message with a few annotations. I've defined a list within the outer function, a list of annotations that I might choose to use. The dash, the asterisk, the plus, etc. Now, I'm going to select one of these annotations at random by invoking the random.choice function. So the annotate variable will hold one of these annotations completely at random. When the code for the outer function greet_with_personal_message is executed, one of these annotations will be chosen at random and stored in the annotate variable.

Now I have a closure defined within this outer function, the closure is named greeting. And within this closure, I have three print statements. I'll first print out the annotations above and below the personal message. And in the middle, I'm going to print out the message and the person to whom the message is addressed, the name. message and name are both passed in as input arguments to the outer function. And finally, I'm going to return a reference to this closure from the outer function, the greet_with_personal_message, so I return greeting. Here is an interesting way you can use closures. I'm going to invoke greet_with_personal_message with different messages and different individuals.

[Video description begins] *She enters a code in the eighteenth code cell. The code is: greet_greg_fn = greet_with_personal_message('Greg', 'Hello!').* [Video description ends]

Here is how I'm going to greet Greg with a Hello. And I store the closure that is returned, which will greet Greg in the greet_greg_fn variable. Then I'm going to invoke greet_with_personal_message once again, but with a different individual and a different message.

[Video description begins] *She enters a code in the nineteenth code cell. The code is: greet_claudia_fn = greet_with_personal_message('Claudia', 'Good morning!').* [Video description ends]

I'm going to greet Claudia with a Good morning. And the closure that is returned with this greeting, I'm going to store in the greet_claudia_fn. The next person that I'm going to grid is Jon with a Hey, and I store this in the greet_jon_fn.

[Video description begins] *She enters a code in the twentieth code cell. The code is: greet_jon_fn = greet_with_personal_message('Jon', 'Hey!').* [Video description ends]

Each invocation of greet_with_personal_message would have made a different random choice of annotations to decorate the message. Whether we use dashes or stars to decorate the message, well, it depends on the random choice that was made. Now let's go ahead and invoke the greet_greg_fn, and you can see that it says Hello Greg. And the Hello Greg message is decorated using the plus sign.

[Video description begins] *She enters a code in the twenty-first code cell. The code is: greet_greg_fn(). She executes the code in the twenty-first code cell. The output reads: Hello! Greg.* [Video description ends]

Observe how the greet_greg_fn has access to the local state that we initialize when this closure was returned. The name was Greg and the message was Hello. Let's invoke the greet_claudia_fn, and you'll see that it says Good morning, Claudia.

[Video description begins] *She enters a code in the twenty-second code cell. The code is: greet_claudia_fn(). She executes the code in the twenty-second code cell. The output reads: Good morning! Claudia.* [Video description ends]

Note that the annotation used to decorate this Good morning Claudia message is different. That's because this was a different random choice for the annotation when we set up the greet_claudia_fn.

[Video description begins] *She enters a code in the twenty-third code cell. The code is: greet_jon_fn(). She executes the code in the twenty-third code cell. The output reads: Hey! Jon.* [Video description ends]

Let's go ahead and invoke our last closure reference here, the greet_jon_fn. And this should say, Hey, Jon. And you can see that it has been randomly decorated using the plus sign. The annotations are picked at random for every invocation of greet_with_personal_message. And each of the individual closure functions remember their own local state. Hello Greg remembers the message Hello and Greg. Good morning Claudia remembers the message Good morning and Claudia. And Hey Jon remembers the message Hey and Jon.

[Video description begins] *She enters a code in the twenty-fourth code cell. The code is: greet_greg_fn(). She executes the code in the twenty-fourth code cell. The output reads: Hello! Greg.* [Video description ends]

Once the local state is associated with a closure, if you invoke the closure once again, it'll use the same state. greet_greg_fn will always say Hello, Greg. And this message will always be decorated using the plus. If you remember, we had mentioned and seen earlier that closures maintain their local state information even when the original function has been deleted.

[Video description begins] *The code of the seventeenth code cell is displayed. She enters a code in the twenty-fifth code cell. The code is, code starts: del greet_with_personal_message greet_with_personal_message('Greg', 'Hello!'). Code ends.* [Video description ends]

I've deleted the greet_with_personal_message function, it can no longer be invoked. As you can see from this error here on screen, greet_with_personal_message is no longer defined.

[Video description begins] *She executes the code in the twenty-sixth code cell. The output displays the following error message: NameError: name 'greet_with_personal_message' is not defined.* [Video description ends]

But greet_greg_fn will still work.

[Video description begins] *The code of the eighteenth, nineteenth, and twentieth code cells is displayed. She enters a code in the twenty-sixth code cell. The code is: greet_greg_fn(). She executes the code in the twenty-sixth code cell. The output reads: Hello! Greg.* [Video description ends]

The closure still exists, and it still has access to the local state. The local state is now associated with the closure itself. greet_claudia_fn will also work, it'll still say Good morning, Claudia.

[Video description begins] *She enters a code in the twenty-seventh code cell. The code is: greet_claudia_fn(). She executes the code in the twenty-seventh code cell. The output reads: Good morning! Claudia.* [Video description ends]

And it still continues to use this little caret symbol to decorate the message. Closures and the local state associated with closures outlive the functions that originally defined the closure. So far, the state that we have associated with the closure has been fairly simple, only string messages, names, etc. In this example, I'm going to associate some more complex state.

[Video description begins] *She enters a code in the twenty-eighth code cell. The code is, code starts: def enroll_in_college(college_name): student_list = [] def enroll_student(student_name): student_list.append(student_name) print("Student", student_name, "has been enrolled in", college_name) print("Current students", student_list, end="\n\n") return enroll_student. Code ends.* [Video description ends]

Here is a function called enroll_in_college, that is the outer function. And the input argument that I pass in here is the name of the college in which we want to enroll a student. Within this outer function, we have a local variable called student_list, which is initially initialized to an empty list. I define a closure within this outer function called enroll_student, which will actually enroll the student in the college. enroll_student takes in one

input argument, student_name. This is the name of the student that we want to enroll in the college specified by college_name.

The first thing we do within enroll_student, that is our closure, is call student_list.append. Any student whom we consider enrolled in that particular college will have his or her name appended to the student_list. Then we'll print out to screen, this student has been enrolled in this college. And here is a current list of students in this college, and we'll print out student_list as well. And finally, in the outer function enroll_in _college, we'll return the enroll_student function object. I'm going to initialize closures to enroll students to a few different colleges.

[Video description begins] *She enters a code in the twenty-ninth code cell. The code is: enroll_in_yale_fn = enroll_in_college('Yale')*. [Video description ends]

Here I want to enroll students in Yale. So the function object that is returned I store in the variable enroll_in_yale_fn. This invocation of the enroll_in_college function will initialize an empty list of students. And that empty list of students will be associated with the enroll_in_yale_fn closure. I'm now going to invoke the enroll_in_college function once again, but this time I want to enroll students into Duke.

[Video description begins] *She enters a code in the thirtieth code cell. The code is: enroll_in_duke_fn = enroll_in_college('Duke')*. [Video description ends]

And the closure that is returned I store in the enroll_in_duke_fn variable. The second invocation of enroll_in_college will create a new empty list of students which is associated with the enroll_in_duke_fn. So we have an entirely new list, empty right now, of students associated with Duke. We are now ready to get started enrolling students.

[Video description begins] *She enters a code in the thirty-first code cell. The code is: enroll_in_yale_fn('Sam').enroll_in_yale_fn('Sam')*. [Video description ends]

Sam has gotten admitted to Yale. So I invoke the enroll_in_yale_fn to enroll Sam to Yale.

[Video description begins] *She executes the code in the thirty-first code cell. The output reads: Student Sam has been enrolled in Yale Current students ['Sam']*. [Video description ends]

This will enroll Sam in Yale, that is, append the name Sam to the list of students associated with the closure which enrolls in Yale. As of now, there is exactly one student who has been enrolled in Yale. Now let's go ahead and invoke the enroll_in_duke_fn, and pass in Jill.

[Video description begins] *She enters a code in the thirty-second code cell. The code is: enroll_in_duke_fn('Jill')*. [Video description ends]

Jill has been admitted to Duke, we'll go and enroll her. You can see that the student_list associated with this closure was previously empty, and now has Jill.

[Video description begins] *She executes the code in the thirty-second code cell. The output reads: Student Jill has been enrolled in Duke Current students ['Sam']*. [Video description ends]

It's pretty clear that the student_list associated with Duke and the student_list associated with Yale are completely different.

[Video description begins] *She enters a code in the thirty-third code cell. The code is, code starts: enroll_in_yale_fn('Donald') enroll_in_yale_fn('John') enroll_in_yale_fn('Trisha'). Code ends*. [Video description ends]

Let's enroll three more students in Yale, Donald, John and Trisha. And the current list of students contains Sam, Donald, John, and Trisha.

[Video description begins] *She executes the code in the thirty-third code cell. The output reads: Student Donald has been enrolled in Yale Current students ['Sam', 'Donald'] Student John has been enrolled in Yale Current students ['Sam', 'Donald', 'John'] Student Trisha has been enrolled in Yale Current students ['Sam', 'Donald', 'John', 'Trisha']*. [Video description ends]

I'll now enroll two more students to Duke University, Olga and Steven are their names.

[Video description begins] *She enters a code in the thirty-fourth code cell. The code is: enroll_in_duke_fn('Olga') enroll_in_duke_fn('Steven')*. [Video description ends]

And Duke currently contains Jill, Olga, and Steven. We used the same function to generate these closures.

[Video description begins] *She executes the code in the thirty-fourth code cell. The output reads: Student Olga has been enrolled in Duke Current students ['Jill, 'Olga'] Student Steven has been enrolled in Duke Current students ['Jill, 'Olga', 'Steven']*. [Video description ends]

But you can see that each closure function has its own local state in different portions of Python memory.

Decorators - Code Modification

[Video description begins] *Topic title: Decorators - Code Modification. The presenter is Janani Ravi*. [Video description ends]

Closures are rather advanced programming constructs, but they do make code elegant and easy to use in certain situations. Closures are also important constructs that are used in Python decorators. What is a decorator, you might ask?

[Video description begins] *She opens the Decorators file in the Jupyter Notebook window. The following message is displayed: Decorators Add functionality to existing code WITHOUT modifying the code*. [Video description ends]

Well, here is a definition that you can chew on before we move on to practical implementation. Decorators help you add functionality to existing code without you having to modify the code itself. Sounds pretty cool, doesn't it? Well, it is indeed cool. Decorators are so called because they decorate code, they don't modify the code, but they make the code do different things using decoration. Now that we've understood closures, we can work our way step by step to understanding and using decorators.

[Video description begins] *She enters a code in the first code cell. The code is, code starts: def print_message(): print("Yoohoo! Decorators are cool!"). Code ends*. [Video description ends]

Let's start off with a simple function which prints a message to screen and simply says, Yoohoo, Decorators are cool.

[Video description begins] *She enters a code in the second code cell. The code is: print_message(). She executes the code in the second code cell. The output reads: Yoohoo! Decorators are cool!*. [Video description ends]

Go ahead and invoke this print message function. And each time you invoke this function it will always print the same thing, Yoohoo, Decorators are cool. Now I feel this is a pretty important message, so when I print it out to screen I want to decorate it with a few highlighters. I want to use a few characters to decorate the original message, and I do this using the highlight function.

[Video description begins] *She enters a code in the third code cell. The code is, code starts: Import random def highlight(): annotations = ['-', '*', '+', ':', '^'] annotate = random.choice(annotations) print(annotate * 50) print_message() print(annotate = 50). Code ends*. [Video description ends]

The highlight function takes in no input arguments, it's an outer function. Within the highlight function, I make a random choice of annotations that I want to use to decorate my original message. I use random.choice to make the random choice between the dash, the asterisk, the plus, the colon, and the caret. Once I've chosen the annotation I want to use to highlight my message, I will print out the annotation character 50 times before and after the message, within that I invoke the print message function.

[Video description begins] *She enters a code in the fourth code cell. The code is: highlight(). She executes the code in the second code cell. The output reads: Yoohoo! Decorators are cool!.* [Video description ends]

All this looks pretty straightforward so far. Let's go ahead and invoke the highlight function, and it will print out our original message, Yoohoo, Decorators are cool, with the random highlight. It happens to be the dash highlight in my case, but in your case it could be something different, it was chosen at random after all. Now let's say we have another message here and I define this message within a function, print another message.

[Video description begins] *She enters a code in the fifth code cell. The code is, code starts: def print_another_message(): print("Did you know! Decorators use closures.”). Code ends.* [Video description ends]

And within that I have the print statement that says, Did you know? Decorators use closures?

[Video description begins] *She enters a code in the sixth code cell. The code is: highlight().* [Video description ends]

Now if I want to highlight this message as well, the existing highlight function won't really help me.

[Video description begins] *She executes the code in the sixth code cell. The output reads: Yoohoo! Decorators are cool!.* [Video description ends]

Because the existing highlight function has been hard coded to invoke the print message function. Because of the hard coding within highlight, it's always going to invoke print message and will always print out, Yoohoo, Decorators are cool, along with the highlight. So how do I change this highlight function so that it's capable of highlighting any message that I want printed out to screen? Well, functions are first class citizens in Python, which means whatever print function you have, you can pass in as an input argument to the highlight function.

[Video description begins] *She enters a code in the seventh code cell. The code is, code starts: def make_highlighted(func): annotations = [‘-‘, ‘*’, ‘+’, ‘:’, ‘^’] annotate = random.choice(annotations) def highlight(): print(annotate * 50) func() print(annotate * 50) return highlight. Code ends.* [Video description ends]

So I'm going to redefine this function, I'll give it a different name as well, I'll call it make highlighted.

[Video description begins] *The code of the third code cell is displayed.* [Video description ends]

The only difference here is that make highlighted takes in an input argument that is a function. This function is what will print out the message I want to display to the screen. This the first change that I have made, what remains the same is how I pick an annotation at random. I have a number of annotations to choose from, and I use the random.choice function to pick one of these. The next change that I have made is that I have defined a closure within the outer make highlighted function, and this is the closure that will perform the actual highlighting. And what is the highlighting? It will simply print out our annotation 50 times before and after the print message. And within this highlight closure, I invoke the function object that was passed in. That is the function object that will actually print out the message, and I return the closure from the make highlighted outer function. Now we have two print functions so far.

[Video description begins] *She enters a code in the eighth code cell. The code is: print_message(). She executes the code in the second code cell. The output reads: Yoohoo! Decorators are cool!.* [Video description ends]

The print message function which says, Yoohoo, Decorators are cool, and a print another message function which says, did you know that decorators use closures?

[Video description begins] *She enters a code in the ninth code cell. The code is: print_another_message(). She executes the code in the second code cell. The output reads: Did you know? Decorators use closures.* [Video description ends]

And now with the help of our make highlighted function, I can highlight any printed message.

[Video description begins] *She enters a code in the tenth code cell. The code is, code starts: highlight_and_print_message = make_highlighted(print_message) highlight_and_print_message(). Code ends.* [Video description ends]

So I'm going to call make highlighted and pass in print message, and then invoke the closure that was returned. Highlight and print message is the closure and it says, Yoohoo, Decorators are cool, and highlights it with some random character.

[Video description begins] *She executes the code in the tenth code cell. The output reads: Yoohoo! Decorators are cool!.* [Video description ends]

Now if I want to print a different message and have it highlighted, I simply pass a different function object in to make highlighted.

[Video description begins] *She enters a code in the eleventh code cell. The code is, code starts: highlight_and_print_another_message = make_highlighted(print_another_message) highlight_and_print_another_message(). Code ends. The code of the seventh code cell is displayed.* [Video description ends]

Here I invoke make highlighted with print another message and whatever closure is returned, I invoke the closure.

[Video description begins] *She executes the code in the eleventh code cell. The output reads: Did you know? Decorators use closures.* [Video description ends]

And that prints out to screen, did you know decorators use closures, with a random character as the highlight. It's pretty clear that the make highlighted function is very generic, you can use it to highlight any message that you want printed to screen. Well, make highlighted is a decorator.

[Video description begins] *The following message is displayed: make_highlighted decorates any message Just pass the message print function as the input argument.* [Video description ends]

Why is it a decorator? Well, it takes in a function object and decorates it and changes it. In our case, it literally highlights it with the character. Decorators are in fact a standard design pattern, and in Python you can use decorators in a really neat manner. Instead of this entire rigmarole of passing in a function object to make highlighted, accessing the closure, and then invoking the closure. You can simply decorate any function, as you see here on screen.

[Video description begins] *She enters a code in the twelfth code cell. The code is, code starts: @make_highlighted def print_a_third_message(): print("Now you'll see how decorators are used"). Code ends.* [Video description ends]

Here, I have a third print message called print a third message, which says, now you'll see how decorators are used. And if I want to highlight it, I simply decorate it using @make highlighted. Remember that the make highlighted function takes in as an input argument the function object. Python will automatically pass in print a third message as the function object, which is the input argument to make highlighted. The use of the decorator

@make highlighted will automatically pass and print the third message as an input to make highlighted and highlight this message. Let's invoke print the third message and you'll see the result.

[Video description begins] *She enters a code in the thirteenth code cell. The code is: print_a_third_message()*. [Video description ends]

The message is automatically highlighted, now you see how decorators are used.

[Video description begins] *She executes the code in the thirteenth code cell. The output reads: Now you'll see how decorators are used*. [Video description ends]

What if you have another message that you need to highlight? Well, specify the decorator @make highlighted. Remember, the decorator is just a method that you've defined previously.

[Video description begins] *She enters a code in the fourteenth code cell. The code is, code starts: @make_highlighted def print_any_message(): print("This is an important result that needs to be highlighted!"). Code ends*. [Video description ends]

That function has to take as an input of function and return a closure.

[Video description begins] *She enters a code in the fourteenth code cell. The code is: print_any_message()*. [Video description ends]

And now if you invoke print any message, you'll find that the result that is displayed to screen is highlighted.

[Video description begins] *She executes the code in the fourteenth code cell. The output reads: This is an important result that needs to be highlighted!*. [Video description ends]

Decorators - Customization

[Video description begins] *Topic title: Decorators - Customization. The presenter is Janani Ravi*. [Video description ends]

Let's see another example of a Decorator, and this decorator won't just be decorative in nature, it will actually do some work.

[Video description begins] *She opens the Decorators file in the Jupyter Notebook window*. [Video description ends]

It will do some error checking for us.

[Video description begins] *She enters a code in the sixteenth code cell. The code is, code starts: import math def area_circle_fn(radius): return math.pi * radius * radius. Code ends*. [Video description ends]

I've imported a math module in Python because I want to use the pi variable within it. I've defined a function here. This is a function that we are familiar with, which calculates the area of a circle given its radius. It uses the formula pi r square.

[Video description begins] *She enters a code in the seventeenth code cell. The code is, code starts: def perimeter_circle_fn(radius): return 2 * math.pi * radius. Code ends*. [Video description ends]

Here is another function definition that calculates the perimeter of the circle given its radius, 2 pi r is the formula.

[Video description begins] *She enters a code in the eighteenth code cell. The code is, code starts: def diameter_circle_fn(radius): return 2 * radius. Code ends*. [Video description ends]

And here is a third function that calculates the diameter of a circle. Once again, the input argument is the radius, 2 into 2 radius is the diameter. In each of these functions, we do no error checking. We assume that the value of radius passed in is positive and correct. I'll first make sure that the functions that we've defined work.

[Video description begins] *She enters a code in the nineteenth code cell. The code is: area_circle_fn(5). She executes the code in the nineteenth code cell. The output reads: 78.53981633974483.* [Video description ends]

I'm going to invoke the area circle of the function. My circle's radius is 5, 78.53 is the area. Well, that's correct, but what if I invoke the area_circle_fn and pass in -1?

[Video description begins] *She enters a code in the twentieth code cell. The code is: area_circle_fn(-1). She executes the code in the nineteenth code cell. The output reads: 3.141592653589793.* [Video description ends]

I'll get a reasonable response, but -1 doesn't really make sense as a value for radius. Our function should have actually thrown an error or told us in some way that negative values of radius are not valid. Now, if you were to perform an error check for each of these functions, we'd have to do it individually. We'd have to have an if statement within the area function, perimeter function as well as the diameter function. That's pretty painful. Instead of that, let's write a decorator that will perform this error checking for us. I'm going to call this safe_calculate, and safe_calculate takes in one input argument that is a function object.

[Video description begins] *She enters a code in the twenty-first code cell. The code is, code starts: def safe_calculate(func): def calculate(r): if r <= 0: raise ValueError("Radius cannot be negative or zero") return func(r) return calculate. Code ends.* [Video description ends]

This is the function object that will actually perform the calculation. Within the safe_calculate outer function, I have an inner function called calculate, this is our closure. calculate takes in one input argument r, that is, the radius. It checks to see whether r is less than or equal to 0. If yes, it throws an error. And the way we throw an error is by simply calling a raise ValueError, Radius cannot be negative or zero. Once we raise this error, Python will stop execution. But if you find that r is positive that's valid, it will invoke the func function and pass in r as an input argument. This is our closure and this is the closure that we returned from the safe_calculate outer function. The safe_calculate is our decorator, which takes as its input a function object and returns a closure that will perform the safe calculation.

[Video description begins] *She enters a code in the twenty-second code cell. The code is, code starts: area_circle_safe = safe_calculate(area_circle_fn). Code ends. The code of the sixteenth code cell is displayed.* [Video description ends]

Before we use it as a decorator, let's just use this safe_calculate manually to ensure that it indeed does work. I invoke safe_calculate, passing the area_circle_fn to calculate the area, store the closure that is return in the variable area_circle_safe. Now, I'll invoke this area_circle_safe closure, passing a value for radius 5.

[Video description begins] *She enters a code in the twenty-third code cell. The code is: area_circle_safe(5).* [Video description ends]

This is safe and I get the result here on screen.

[Video description begins] *She executes the code in the twenty-third code cell. The output reads: 3.141592653589793.* [Video description ends]

If I try to invoke this area_circle_safe closure with a negative value for radius, that's when I encounter an error.

[Video description begins] *She enters a code in the twenty-fourth code cell. The code is: area_circle_safe(-1).* [Video description ends]

Within our closure, we check whether r is less than equal to 0 and if it is, we raise a ValueError which says, Radius cannot be negative or zero.

[Video description begins] *She executes the code in the twenty-fourth code cell. The output displays the following error message: ValueError: Radius cannot be negative or zero. The code of the the twenty-first code cell is displayed.* [Video description ends]

Let's decorate the perimeter function as well with the safe_calculate.

[Video description begins] *She enters a code in the twenty-fifth code cell. The code is, code starts: perimeter_circle_safe = safe_calculate(perimeter_circle_fn) perimeter_circle_safe(10). Code ends. The code of the seventeenth code cell is displayed.* [Video description ends]

Invoke safe_calculate, pass in the perimeter_circle function, we get a closure at its return. Invoke the closure stored in the perimeter_circle_safe variable. Let's pass in a radius of 10. This is completely safe, and you get the result displayed on screen.

[Video description begins] *She executes the code in the twenty-fifth code cell. The output reads: 62.83185307179586.* [Video description ends]

But if you were to call perimeter_circle_safe with a negative value for radius, well, that is a ValueError.

[Video description begins] *She enters a code in the twenty-sixth code cell. The code is: perimeter_circle_safe(-10). She executes the code in the twenty-sixth code cell. The output displays the following error message: ValueError: Radius cannot be negative or zero.* [Video description ends]

Now that you have a decorator, you should decorate your functions rather than use the long winded way that we've been using so far.

[Video description begins] *She enters a code in the twenty-seventh code cell. The code is, code starts: @safe_calculate def area_circle_fn(radius): return math.pi * radius * radius @safe_calculate def perimeter_circle_fn(radius): return 2 * math.pi * radius. Code ends.* [Video description ends]

Here we have our area_circle_fn and perimeter_circle_fn redefined. I've decorated both of these with the safe_calculate decorator. Things are intuitive and simple this way. I don't need to remember and store the returned closure in a variable. Now, the next time I call the area_circle_fn or the perimeter_circle_fn, the safe check will be performed.

[Video description begins] *She enters a code in the twenty-eighth code cell. The code is: perimeter_circle_fn(3). She executes the code in the twenty-eighth code cell. The output reads: 18.84955592153876.* [Video description ends]

The perimeter of a positive value for radius is valid and displayed on screen. If you try to calculate the perimeter for a negative value of radius, well, you'll get a ValueError. The safe_calculate function that we set up earlier has a limitation, and you'll see what it is in just a bit.

[Video description begins] *She enters a code in the twenty-ninth code cell. The code is: perimeter_circle_fn(-3). She executes the code in the twenty-ninth code cell. The output displays the following error message: ValueError: Radius cannot be negative or zero.* [Video description ends]

Here is a function which calculates the area of a rectangle.

[Video description begins] *She enters a code in the thirtieth code cell. The code is, code starts: @safe_calculate def area_rectangle_fn(length, breadth): return length * breadth. Code ends.* [Video description ends]

Remember, to calculate the area of a rectangle, you need to pass in two input arguments, length as well as breadth of the rectangle. But within our safe_calculate function, we had invoked the func object which actually performs the calculation with just one input argument, with just the variable r.

[Video description begins] *The code of the twenty-first code cell is displayed*. [Video description ends]

And this is going to cause a problem when we use the safe_calculate decorator for the area rectangle function.

[Video description begins] *She enters a code in the thirty-first code cell. The code is: area_rectangle_fn(4, 5)*. [Video description ends]

Let's see how. Once I have decorated this function, I'm going to invoke it with 4, 5.

[Video description begins] *She executes the code in the thirty-first code cell. The output displays the following error message: TypeError: calculate() takes 1 positional argument but 2 were given*. [Video description ends]

Immediately, I get a TypeError which says calculate takes 1 positional argument but 2 were given. The problem is with the way we had defined the closure inside the safe_calculate function.

[Video description begins] *She points to the code of the twenty-first code cell*. [Video description ends]

The calculate closure takes in just one input argument. So if you have a function with multiple input arguments, then you can't decorate it using safe_calculate. Well, if I want a safe_calculate that works with any number of input arguments, I need to make a few changes in my safe_calculate function. So I'm going to redefine it and call it safe_calculate_all.

[Video description begins] *She enters a code in the thirty-second code cell. The code is, code starts: def safe_calculate_all(func): def calculate(*args): for arg in args: if arg <= 0: raise ValueError("Arguments cannot be negative or zero") return func(*args) return calculate. Code ends*. [Video description ends]

It takes in one input argument that is the function object that is to be decorated. The main change that I have made is in the input arguments that I pass in to the calculate closure. calculate now takes in variable length arguments, *args. And within calculate, I iterate over all of the arguments that were passed in, and check to see whether the argument is less than or equal to 0. If any of the arguments are less than or equal to 0, I'll raise a ValueError. If all of the arguments are positive, then I'll invoke the function object that was passed in, and pass in *args to the function. Remember, *args will unpack the original arguments so that the elements of the tuple are passed in individually to the function object, func. You can now use this safe_calculate_all decorator with functions that have any number of arguments.

[Video description begins] *She enters a code in the thirty-third code cell. The code is, code starts: @safe_calculate_all def area_rectangle_fn(length, breadth): return length * breadth @safe_calculate_all def perimeter_rectangle_fn(length, breadth): return 2 * (length + breadth). Code ends. The code of the thirty-second code cell is displayed*. [Video description ends]

I'm going to use safe_calculate_all to decorate the area_rectangle_function as well as a perimeter_rectangle_function. Both of these functions take in to input arguments, the length and the breadth of the rectangle. Thanks to the fact that the calculate closure takes in a variable length of arguments, you can pass in any number of arguments. And safe_calculate_all will decorate all kinds of functions.

[Video description begins] *She enters a code in the thirty-fourth code cell. The code is: area_rectangle_fn(5, 4)*. [Video description ends]

Let's invoke the area rectangle function with 5 and 4, both of these are positives, so we get the area 20.

[Video description begins] *She executes the code in the thirty-fourth code cell. The output reads: 20*. [Video description ends]

Let's try invoking the same function, but this time I'll make one of the arguments negative, the breadth is negative and that gives me a ValueError, thanks to our safe_calculate_all decorator.

[Video description begins] *She enters a code in the thirty-fifth code cell. The code is: area_rectangle_fn(5, -4).* [Video description ends]

When you invoke this malfunction, it'll check all arguments.

[Video description begins] *She executes the code in the thirty-fifth code cell. The output displays the following error message: ValueError: Arguments cannot be negative or zero.* [Video description ends]

It doesn't matter which argument is negative, you still get the ValueError. Here I pass the length as negative, and I still get the ValueError.

[Video description begins] *She enters a code in the thirty-sixth code cell. The code is: area_rectangle_fn(-5, 4). She executes the code in the thirty-sixth code cell. The output displays the following error message: ValueError: Arguments cannot be negative or zero. The code of the thirty-third code cell is displayed.* [Video description ends]

Let's try the parimeter_rectangle_fn. When both positive arguments are present, I get the perimeter of this rectangle.

[Video description begins] *She enters a code in the thirty-seventh code cell. The code is: perimeter_rectangle_fn(5, 4). She executes the code in the thirty-seventh code cell. The output reads: 18.* [Video description ends]

Let's make one of these arguments negative. I pass in -5 and that throws a ValueError.

[Video description begins] *She enters a code in the thirty-eighth code cell. The code is: perimeter_rectangle_fn(-5, 4). She executes the code in the thirty-seventh code cell. The output displays the following error message: ValueError: Arguments cannot be negative or zero. The code of the thirty-third code cell is displayed.* [Video description ends]

The safe_calculate_all will work with function with any number of arguments.

[Video description begins] *She enters a code in the thirty-ninth code cell. The code is, code starts: @safe_calculate_all def perimeter_circle_fn(radius): return 2 * math.pi * radius. Code ends.* [Video description ends]

You can use it to decorate the perimeters of the function that takes in just one argument and it'll work.

[Video description begins] *She enters a code in the fortieth code cell. The code is: perimeter_circle_fn(4). She executes the code in the fortieth code cell. The output reads: 25.132741228718345.* [Video description ends]

Let's invoke this perimeter_circle_function with a positive radius, well, that should calculate the perimeter just fine.

[Video description begins] *She enters a code in the forty-first code cell. The code is: perimeter_circle_fn(-4). She executes the code in the fortieth code cell. The output reads displays the following error message: ValueError: Arguments cannot be negative or zero.* [Video description ends]

Let's invoke it with the negative radius and this should result in a ValueError, and it does.

Chaining Decorators

[Video description begins] *Topic title: Chaining Decorators. The presenter is Janani Ravi.* [Video description ends]

Now in Python, it's not really necessary that you use just one decorator with a function.

[Video description begins] *She opens the Decorators file in the Jupyter Notebook window.* [Video description ends]

You can have a function decorated using multiple decorators. And these decorators will be chained together.

[Video description begins] *She enters a code in the forty-second code cell. The code is, code starts: def asterisk_highlight(func): def highlight(): print("*" * 50) func() print ("*" * 50) return highlight. Code ends.* [Video description ends]

I'm going to define a simple decorator for a printed message here which simply highlights that printed message using the asterisk. Think of this as the asterisk decorator. It takes as its only input argument a function object. Within that we define a closure called highlight, this highlight closure prints out asterisk before and after invoking the function object. And this highlight closure is what we return from this decorator. I'll now define another decorator function called plus_highlight.

[Video description begins] *She enters a code in the forty-third code cell. The code is, code starts: def plus_highlight(func): def highlight(): print("+" * 50) func() print ("+" * 50) return highlight. Code ends.* [Video description ends]

It takes as its only input argument a function object. The highlight closure within this plus highlight outer function prints out the plus character to screen before and after invoking the function represented by the func object. And this highlight closure is what is returned from the plus_highlight function as well. We have two decorators, let's first use the asterisk_highlight decorator to decorate print message one.

[Video description begins] *She enters a code in the forty-fourth code cell. The code is, code starts: @asterisk_highlight def print_message_one(): print("Yoohoo! Decorators are cool!"). Code ends.* [Video description ends]

The message that this function prints is simply Yoohoo, decorators are cool.

[Video description begins] *She enters a code in the forty-fifth code cell. The code is: print_message_one().* [Video description ends]

And this message will be highlighted using the asterisk, invoke print message one, and you'll see that the message is highlighted using the asterisk as we expected.

[Video description begins] *She executes the code in forty-fifth code cell. The output reads: Yoohoo! Decorators are cool. The code of the the forty-third code cell is displayed.* [Video description ends]

Now let's define another print function, but this time we'll decorate it using two decorators, the plus_highlight and the asterisk_highlight.

[Video description begins] *She enters a code in the forty-sixth code cell. The code is, code starts: @plus_highlight @asterisk_highlight def print_message_one(): print("Yoohoo! Decorators are cool!). Code ends.* [Video description ends]

What you see here is an example of chaining decorators together. But how are they chained? Which decoration comes first, the asterisk_highlight or the plus_highlight? Whichever decorator is the closest to the function definition is what is executed first, and then the decorator which is further away from the function definition. This means our message here, Yoohoo, decorators are cool, will be first highlighted with the asterisk, then the plus.

[Video description begins] *She enters a code in the forty-seventh code cell. The code is: print_message_one(). She executes the code in forty-seventh code cell. The output reads: Yoohoo! Decorators are cool.* [Video

description ends]

Let's try this, go ahead and invoke print message one, and you can see Yoohoo, decorators are cool is at the very center.

[Video description begins] *She points to the code of the forty-second code cell*. [Video description ends]

Just outside it we have the asterisk highlight, and the outermost highlight is the plus highlight.

[Video description begins] *She points to the code of the forty-third code cell*. [Video description ends]

If you change the order of the decorators, the decorations order will change as well.

[Video description begins] *She enters a code in the forty-eight code cell. The code is, code starts: @asterisk_highlight @plus_highlight def print_message_one(): print("Yoohoo! Decorators are cool!). Code ends*. [Video description ends]

You'll have the same function print message one, but the decorator that is closest to the function definition is the plus_highlight and then the asterisk_highlight.

[Video description begins] *She enters a code in the forty-ninth code cell. The code is: print_message_one(). She executes the code in forty-ninth code cell. The output reads: Yoohoo! Decorators are cool*. [Video description ends]

So if you invoke print message one, the message will first be highlighted using the plus, then the asterisk.

[Video description begins] *She points to the code of the forty-third code cell*. [Video description ends]

You have successfully seen how we can chain decorators, and we've also understood the order in which the chained decorators are applied.

[Video description begins] *She points to the code of the forty-second code cell*. [Video description ends]

The one closest to the function definition is applied first, and then we move outward.

Exercise: Advanced Features in Python Functions

[Video description begins] *Topic title: Exercise: Advanced Features in Python Functions. The presenter is Janani Ravi*. [Video description ends]

At this point in time, you should have a good, comfortable working knowledge of the advanced features that Python functions offer. So let's move on to exercises. Here, try and specify the details that you need to keep in mind while writing a recursive function. Then try and define what a generator function is, how it works and how it is different from a regular function. And finally, you will try and list the characteristics and use cases of closures in Python. These are a few interesting points for you to ponder over. I suggest you try and mull over them. Answer these questions on your own before we move on to a sample solution. Now would be a good time for you to pause your video for a bit. Let's begin our discussion with recursive functions.

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

We've seen that a function within its body can invoke any other function, whether it's a built-in function or a custom function. A function can also invoke itself and that is what a recursive function is. When you work with recursive functions in Python, you'll see that there is a limit for how many times a function can call itself. This is a system limit and this is a limit that you can configure. The default limit when we work with Python was 3,000, that is, a function can call itself 3,000 times, but this is something that you can change. Now when you have a

recursive function call itself, this invocation over and over again can kind of get into an infinite loop-like situation. So when you're writing a recursive function, the first thing that you need to think about is when will this function terminate? You need to specify a terminating condition so that the function no longer recurses. In all other ways, your recursive function is just like an ordinary function.

It may take an input argument, it may or may not return values. We'll now move on to discussing generator functions in Python. A generator function is a special kind of function that allows you to generate an infinite sequence that can be iterated over. An iterator object in Python is any object that responds to the built-in next function to give you the next element in a sequence. A generator function allows you to create an iterator which allows iteration over elements in a sequence. If you want to generate very long sequences, generator functions are preferred. Because they are a memory efficient way to iterate over infinite sequences or very long sequences. All of the elements of the sequence need not be present in memory at the same time. This is what makes a generator function memory efficient. A generator function does not use the return statement. Instead, it uses the yield command within the function body to yield control back to the calling program. Invoking the generator function does not execute the code of the generator function unlike with regular functions. Regular functions are executed when invoked.

A generator function, when invoked, returns a generator object. And this generator object is what we'll use to iterate over a sequence. The code within a generator function is executed when the next() method is called on the generator object, or when you use the generator with a for loop which automatically calls this next method. In a regular function, the state of the local variables are not maintained across function calls. In the case of a generator function, the generator remembers the state of the local variables across function invocations. And finally, let's move on to discussing what closures are and where they might be used. A function nested within another function is called a closure. When the definition of the function is within another outer function, that inner function is called a closure. A closure has access and can refer to local variables that have been defined in the outer function.

Input arguments that are passed in to the outer function, local variables that have been declared in the outer function, the closure has access to all of these. Now, the cool thing about closures is the fact that this access to the local variables persists even when the outer function is no longer present in Python memory. The local state is associated with the closure itself. Even when the outer function goes out of memory or you delete it, the closure can still access this local state. So you can think of closures as carrying around local state information along with the closure definition. Closures are extremely important concepts used to implement something called decorators in Python. Decorators in Python are used to add functionality to code without modifying the code itself. So decorators are very popular in Python and closures are what we use to implement decorators.