Data Structures & Algorithms in Python: Implementing Data Structures

Examine operations that have different values of time complexity and delve into implementation of basic data structures, such as linked lists, stacks, and queues in Python, in this 13-video course. Key concepts covered here include operations that run in constant time regardless of input; code whose time complexity varies directly with value of input; and tasks whose time complexity varies linearly with size of input. Next, you will learn about operations whose time complexity varies as the square of input size; how to use native queue class of Python and perform standard queue operations; and how to code a queue class for many standard queue operations, such as enqueue and dequeue. Then, learn how a Python list can be used as a stack by loading and unloading elements, and how to implement a custom stack class for common stack operations. Finally, study code functions to perform search and delete operations in linked lists and reverse the ordering of its nodes; and create a linked list and test out various operations that have been defined.

Table of Contents

Course Overview

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

Hi and welcome to this course, Implementing Data Structures in Python. My name is Kishan Iyer and I will be your instructor for this course. A little about myself first, I have a master's degree in Computer Science from Columbia University, and have previously worked in companies such as Deutsche Bank and WebMD in New York. I presently work for Loonycorn, a studio for high-quality video content. Data structures and algorithms are the bread and butter of coding. Any programmer will use these every single day. And a good programmer will have an in depth understanding of the various data structures and algorithms which can be applied in different situations. Data structures and algorithms go hand in hand, and which ones you pick determine not only the kinds of operations which you can perform but also the speed with which they are executed and the amount of compute resources which they consume.

This course is very much hands on and contains only labs, where we use Python 3 as our language and Jupyter notebooks as our IDE. We start off by performing a number of basic tasks using functions and loops and examine the complexities of these operations. We then move on to queue data structures, where we use the native queue class and Python, and also implement our own custom queue. We explore how a Python list can be used as a stack and then define a bespoke stack class. And finally, we construct our own link list, which includes implementations of all the basic link list operations, plus a few others as well. By the end of this course, you will be quite the expert on link lists, stacks, and queues. You will have an in depth understanding of the different operations on each of these structures and how they can be implemented in Python.

O(1) Operations

[Video description begins] *Topic title: O(1) Operations. The presenter is Kishan Iyer.* [Video description ends]

In this course, we will be writing Python code for several data structure operations. We will examine how these operations can be implemented, and we'll also take a look at the complexity of those operations along various measures. All the code in this course will be written using Jupyter notebooks as the IDE. Jupyter notebooks, in case you haven't used it before, is an interactive web-based IDE which is very popular among data scientists. One reason for this is that it allows you to execute chunks of code and view the results of the execution before proceeding with the development of your program.

[Video description begins] *He opens the Jupyter Notebook window. It is divided into two parts. The first part includes Quit and Logout buttons. The second part contains three panes labeled: Files, Running, and Clusters. The Files pane is selected. It is divided into three sections. The first section includes an Upload button and a New drop-down button. The second section includes Last Modified and File size buttons. The third section is the content pane.* [Video description ends]

In order to install Jupyter notebooks, you can head to the link which you see on the screen in front of you, and then follow the instruction which are relevant for your own environment.

[Video description begins] *The link is: https://jupyter.org/install.* [Video description ends]

Once you do that, you can navigate your workspace directory and then bring up the Jupyter notebook IDE from there, which is what I have done. And now I'm going to create a new Jupyter notebook for our first demo.

[Video description begins] *He clicks the New drop-down button, a drop-down list opens, which includes Python 3, Text File, and Folder options. He selects the Python 3 option.* [Video description ends]

So I navigate over to the New button, and then choose the Python 3 kernel.

[Video description begins] *An untitled notebook opens. It is divided into four parts. The first part contains a Logout button. The second part is the menu bar, which includes File, Edit, View, and Insert menus. The third part is the toolbar, which includes Run and Code buttons. The fourth part is the content pane.* [Video description ends]

Once that is done, the first thing I'm going to do is to rename this notebook, which has been given a default title of Untitled, and then give it a name which is more representative of what we are about to do.

[Video description begins] *The Rename Notebook dialog box opens. It contains Enter a new notebook name text box and Cancel and Rename buttons.* [Video description ends]

In this first demo, we will be studying the time complexity of various operations. So this is what I'm going to set the name of this notebook to.

[Video description begins] *He types Time Complexity in the Enter a new notebook name text book and clicks the Rename button. The Rename Notebook dialog box closes.* [Video description ends]

So what is the first operation for which we will study the time complexity? Well, we will now write a function in order to perform a simple addition of two numbers.

[Video description begins] *He enters a code in the first code cell. The code is, code starts: def addition(num1, num2): total = num1 + num2 print("The sum of %d and %d is %d" % (num1, num2, total)). Code ends.* [Video description ends]

So I'm going to create a Python function called addition, this takes in two arguments, num1 and num2. Within the body of the function, we simply sum up these two numbers by performing an addition, and this is assigned to a variable total. And then we include a print statement where we print out both of the numbers which were passed as argument, along with the sum of those numbers. All right, so this is about as simple as it can get in terms of operations.

So we just run the cell. And in case you're not familiar with the Jupyter notebook environment, you can do this by first selecting the cell and hitting the Run button at the top. Or if you're on a macOS, you can hit Shift+Enter or Ctrl+Enter. Once that is done, the cell has been executed, in which case a function definition is now in place. It is now time for us to invoke this function, and we will just sum up the numbers of 10 and 4.

[Video description begins] *He enters a code in the second code cell. The code is: addition(10,4).* [Video description ends]

On running the cell, the print statement has been executed, and we correctly get the sum of the two numbers as well.

[Video description begins] *He executes the code in the second code cell. The output reads: The sum of 10 and 4 is 14.* [Video description ends]

All right, so our addition function has been defined, but what exactly is the time complexity of this operation? For that, we're not exactly interested in the number of steps involved within the function definition. Rather, we are more interested in the total number of iterations which any addition operation involves. To demonstrate that, we redefine our addition function with a couple of minor alterations.

[Video description begins] *He enters a code in the third code cell. The code is, code starts: def addition(num1, num2): num_iterations = 0 total = num1 + num2 num_iterations += 1 print("The sum of %d and %d is %d \nTotal number of iterations = %d" %(num1, num2, total, num_iterations)). Code ends.* [Video description ends]

The function has the same name and also accepts the same arguments. But in this occasion, we introduce a variable called num_iterations, which we initialize to 0. Following that, we perform our total calculation just as we did before, and then increment the num_iterations variable. And then include a print statement, similar to what we had previously, which includes the arguments and the total, but also put in the number of iterations here.

[Video description begins] *He enters a code in the fourth code cell. The code is: addition(10,4).* [Video description ends]

With this modified addition function, we will execute this cell, and then head over and call this function once more with the same arguments which we used previously. And as you would expect, the total number of iterations is 1.

[Video description begins] *He executes the code in the fourth code cell. The output reads: The sum of 10 and 4 is 14 Total number of iterations = 1.* [Video description ends]

So what about the time complexity of this addition operation? Well, as we covered in the theory section of this learning path, the time complexity of any operation or algorithm is a representation of how the number of iterations or individual operations were to change with the contents of the input. So to demonstrate that, we will now change the input. So on this occasion, we call the addition function by specifying the values of 10000 and 200 as the arguments.

[Video description begins] *He enters a code in the fifth code cell. The code is: addition(10000, 200).* [Video description ends]

And when we run this, once again, we notice that the number of iterations is exactly 1.

[Video description begins] *He executes the code in the fifth code cell. The output reads: The sum of 10000 and 200 is 10200 Total number of iterations = 1.* [Video description ends]

You will know that no matter what arguments we supply to this function, the number of iterations will continue to remain 1.

[Video description begins] *The following message is displayed: Number of iterations is always 1- this operation runs in constant time.* [Video description ends]

So the number of operations involved will be constant no matter what the input. In other words, this operation runs in constant time, which means that the complexity can be represented as Big O (1). It is now time to move on to one more function which represents another set of operations. So this is a function called check_oddeven, which, as the name implies, will accept an argument and will check whether that number happens to be an odd or even number.

[Video description begins] *He enters a code in the sixth code cell. The code is, code starts: def check_oddeven(number): count = 0 num_iterations = 0 if (number%2 == 0): num_iterations += 1 print("%d is an even number" % number) else: num_iterations += 1 print("%d is an odd number" % number) print("Total number of iteration = %d " %(num_iterations)). Code ends.* [Video description ends]

So once we define this function signature, we use two variables, one is called count and the other is called number of iterations, both of which are initialized to 0. You will notice within the function body that account variable is not referenced anymore. In just a little bit, I'll get to why exactly this is the case. So within the function body, we start off by performing a modulus 2 operation on our input. And then if this number is divisible by 2, we increase the value of num_iterations by 1. And then we print out the fact that this argument is an even number.

If the mod 2 operation does not give us zero, in which case it is not divisible by 2, we increment the value of num_iterations once more and then print out the fact that this is an odd number. And then at the very end of this function, we just print out the value of num_iterations. All right, so when compared to the previous addition function, we see that this particular function involves the execution of more lines of code. So we will now run the cell and then call this function, so we will check whether the number 5 is an odd or even function.

[Video description begins] *He enters a code in the seventh code cell. The code is: check_oddeven(5).* [Video description ends]

And then we observe that this is indeed an odd number, of course, and that the total number of iterations is 1.

[Video description begins] *He executes the code in the seventh code cell. The output reads: 5 is an odd number Total number of iteration = 1.* [Video description ends]

We will now call the function once more with an argument of 2500.

[Video description begins] *He enters a code in the eighth code cell. The code is: check_oddeven(2500).* [Video description ends]

And once again, the total number of iterations is 1.

[Video description begins] *He executes the code in the eighth code cell. The output reads: 2500 is an even number Total number of iteration = 1.* [Video description ends]

So no matter what the input, the total number of iterations is constant.

[Video description begins] *The following message is displayed: This is also a constant time operation*. [Video description ends]

Now, this is important because even though there are more lines of code which are executed within this check_oddeven function when compared to our addition function. It is the total number of iterations which we are concerned with when determining the complexity of an operation. So even though this function definition included a line of code initializing the count variable which is totally unnecessary, and since the number of iterations is constant no matter what the input. We say that this check_oddeven operation also executes in constant time, that is, the time complexity is Big O (1). Moving along now to one more function, which also executes in constant time. And this is called find_square, which will return the square of an input number.

[Video description begins] *He enters a code in the ninth code cell. The code is, code starts: def find_square(number): num_iterations = 0 square = number**2 num_iterations += 1 print("Square of %d is %d \nTotal number of iteration = %d " % (number, square, num_iterations)). Code ends*. [Video description ends]

We once again initialize a num_iterations variable, and this is initially set to 0. And then we calculate the square as the number to the power of 2. We then increment the value of num_iterations, and then include a print statement, which prints out the input, the square, as well as the number of iterations.

[Video description begins] *He enters a code in the tenth code cell. The code is: find_square(2)*. [Video description ends]

And then when we run the find_square function, this returns us the square of 2, along with the fact that the number of iteration was 1.

[Video description begins] *He executes the code in the tenth code cell. The output reads: Square of 2 is 4 Total number of iteration = 1*. [Video description ends]

The number of iterations will remain 1 if we input a larger argument to find_square, and this is confirmed by running the cell.

[Video description begins] *He enters a code in the eleventh code cell. The code is: find_square(10000)*. [Video description ends]

So this is what we mean by a constant time operation, where the total number of iterations is constant no matter what the input.

[Video description begins] *He executes the code in the eleventh code cell. The output reads: Square of 10000 is 100000000 Total number of iteration = 1*. [Video description ends]

But this is, of course, independent of what exactly goes on within the operations. This means that even though two operations may be set to execute in constant time, it does not mean that they execute the same number of steps. All it means is that the number of steps does not vary with the input.

O(n) Operations - Part 1

[Video description begins] *Topic title: O(n) Operations - Part 1. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we took a look at a few different operations which execute in constant time. We will now explore some other operations where the number of iterations varies linearly with the input.

[Video description begins] *He opens the Time Complexity file in the Jupyter Notebook window*. [Video description ends]

The first of the functions we will now take a look at if one to check whether a particular number happens to be a prime number. We will be using two different variations of this function. So this is called check_prime1.

[Video description begins] *He enters a code in the twelfth code cell. The code is, code starts: def check_prime1(number): num_iterations = 0 for i in range(2, number): num_iterations += 1 if number%i == 0: print("%d is not a prime number \nTotal number of iterations = %d" %(number, num_iterations)) return print("%d is a prime number \nTotal number of iterations = %d" %(number, num_iterations)). Code ends*. [Video description ends]

This of course accepts some integer as an argument, and then within the function body, we initialize a variable called num_iterations and set it to 0. And then, following that, we include a for loop where we will iterate through all of the numbers between 2 and 1 less than our input. And then, check whether our input happens to be divisible by one of those numbers. So for each iteration of the for loop, we increment the value of num_iterations variable. And then, we will perform a modulus operation to check whether the argument to a function happens to be divisible by i. If yes, we will simply print out the fact that this particular argument is not a prime number.

And then, we also return the number of iterations which have been carried out to come to this conclusion. We will then exit out of this function by running return, and if we go through the entire range of values for the for loop and still don't find a number which our argument is divisible by, then we will print out the fact that this argument is a prime number. Once again, we will print out the number of iterations required to come to this conclusion. All right, so with a function definition complete, we run the cell and then call our function.

[Video description begins] *He enters a code in the thirteenth code cell. The code is: check_prime1(1)*. [Video description ends]

And for the first execution, we will check whether the number 1 can be considered a prime.

[Video description begins] *He executes the code in the thirteenth code cell. The output reads: 1 is a prime number Total number of iterations = 0*. [Video description ends]

And based on the logic which we have defined, the number 1 is indeed a prime number but the total number of iterations of the for loop is 0. This is because the range of the for loop begins at the number 2. We will now change our argument and then call our function once more with an argument of 49.

[Video description begins] *He enters a code in the fourteenth code cell. The code is: check_prime1(49)*. [Video description ends]

And we know that the lowest common factor for this number is 7, which is why when we run this, it takes a total of six iterations to find a number which 49 is divisible by, specifically 7.

[Video description begins] *He executes the code in the fourteenth code cell. The output reads: 49 is not a prime number Total number of iterations = 6*. [Video description ends]

All right, so we have seen that by changing the value of the argument, the number of iterations in our function does change. So what if we were to change the argument once more? This time, we will pass in a prime number which is 193, and then call a check_prime1 function. So this prints out that 193 is indeed a prime number, but the total number of iterations is 191. This is because the for loop executed for each integer between 2 and 190. We can now run this function once more for a much larger prime number which is 1,847.

[Video description begins] *He enters a code in the sixteenth code cell. The code is: check_prime1(1847). He executes the code in the sixteenth code cell. The output reads: 1847 is a prime number Total number of iterations = 1845*. [Video description ends]

And on this occasion, the total number of iterations required to detect the fact that this is a prime number is 1,845. So once again, this is just two less than the number itself. So we have now seen that this check_prime1 function in the worst case requires pretty much as many iterations as the value of the number. So if our input happens to be n, then the number of iterations require is also proportional to n. Which is why we can say that the complexity of this particular operation, which is to check whether the number is prime or not, is Big O (n). Now, what if we were to modify this check_prime function just a little bit? So previously, we iterated through all of the values between 2 until 1 less than our input argument in order to find whether the number is a prime. But this is not the most efficient way to check for a prime number.

[Video description begins] *He enters a code in the seventeenth code cell. The code is, code starts: def check_prime2(number): num_iterations = 0 mid_point = int(number / 2) for i in range(2, mid_point): num_iterations += 1 if number%i == 0: print("%d is not a prime number \nTotal number of iterations = %d" %(number, num_iterations)) return print("%d is a prime number \nTotal number of iterations = %d" %(number, num_iterations)). Code ends.* [Video description ends]

We will now use a method which is a little more efficient in the function check_prime2 where we once again start off by setting the num_iterations variable to 0. But we will now reduce the number of iterations to effectively half of what we saw in the previous version of this function. This is because we know that any number n cannot really be divisible by any other number between half of n and n minus 1. So we will now check whether our input argument is divisible by any number between 2 and half of the argument. So we calculate this mid_point right here. And then, following that, we use that midpoint value in our for loop.

So we iterate between 2 and this midpoint value. So you can already imagine that, in the worst case, the total number of iterations will effectively be half of what we saw in the previous version of our check_prime function. But once again, we keep a count of the number of iterations which is incremented in each iteration of the for loop. We perform a divisibility check once more. And if our input happens to be divisible by one of the numbers, then we would print out the fact that it's not a prime.

[Video description begins] *He enters the following code in the eighteenth code cell: check_prime2(1).* [Video description ends]

And then, again, if you go through the entire for loop without finding a divisor, we will just print out the fact that this is a prime number. So with this modified version of check_prime defined, we will run the cell and go ahead and invoke this function. The first argument we pass is 1.

[Video description begins] *He executes the code in the eighteenth code cell. The output reads: 1 is a prime number Total number of iterations = 0.* [Video description ends]

And just as we saw in the previous version of the function, the total number of iterations of the for loop is 0.

[Video description begins] *He enters the following code in the nineteenth code cell: check_prime2(49).* [Video description ends]

When we call the function once again with an argument of 49, just as before, the total number of iterations is 6.

[Video description begins] *He executes the code in the nineteenth code cell. The output reads: 49 is not a prime number Total number of iterations = 6.* [Video description ends]

So far, our modifications to the function don't seem to have had an effect on the total number of operations.

[Video description begins] *He enters a code in the twentieth code cell. The code is: check_prime2(193).* [Video description ends]

However, when we invoke this function for a prime number, then we find that the total number of iterations has approximately halved.

[Video description begins] *He executes the code in the twentieth code cell. The output reads: 193 is a prime number Total number of iterations = 94.* [Video description ends]

So we have now concluded that 193 is a prime number in 94 iterations rather than 191. So we have checked whether 193 is divisible by any number between 2 and 95 rather than by any number between 2 and 192. All right, what if we run check_prime2 for 1,847? And once again, the total number of iteration has approximately halved to about 921. So if the previous version of our check_prime function had a time complexity of Big O (n), what about this version? Well, this also has a complexity of Big O (n).

So even though the total number of iterations required has halved in the worst case, the number of iteration is still directly proportional to the value of the input n. This is an important factor to keep in mind when working with the Big O notation where it may not be granular enough to capture the fact that one algorithm runs in half the time as another. This is because the Big O notation wishes to capture bigger swings in the scale of operations require. Such as a quadratic or cubic relationship between the input and the number of iterations.

O(n) Operations - Part 2

[Video description begins] *Topic title: O(n) Operations - Part 2. The presenter is Kishan Iyer.* [Video description ends]

We now move along to a function whose argument happens to be a list of values rather than an individual integer.

[Video description begins] *He opens the Time Complexity file in the Jupyter Notebook window.* [Video description ends]

This is a little more representative of the type of operation for which a complexity is calculated, because we are more interested in how the number of operations varies based on the size of the input rather than on the value of the input. So here, we have a function called find_maxval, which will iterate over a list, and then find the maximum value within that list.

[Video description begins] *He enters a code in the twenty-second code cell. The code is, code starts: def find_maxval(num_list): maxval = num_list[0] num_iterations = 0 for i in range(len(num_list)): num_iterations += 1 if maxval < num_list[i]: maxval = num_list[i] print("Maximum value of the list = %d \nTotal number of iterations = %d" %(maxval, num_iterations)). Code ends.* [Video description ends]

So here, we initially set the maxval to be the first element within the input list, and then we also initialize the number of iterations to 0. We then make use of a for loop, which iterates through every single element within the list. And then in each iteration of the for loop, we increment the value of num_iterations, and then we compare the value of maxval with that particular element within the list. And then if it happens to be greater, we update the value of maxval. And finally, we print out the total number of iterations which we needed to go through, and the maximum value within our list. So with our function defined, we run the cell, and then we invoke this function on this list of four integers.

[Video description begins] *He enters a code in the twenty-third code cell. The code is: find_maxval([123, 67, 25, 79]).* [Video description ends]

So the maximum value has been returned as 123 as we should expect, and the total number of iterations is equal to the size of the list which is 4. We now run find_maxval for a second time, with a list of six integers.

[Video description begins] *He enters a code in the twenty-fourth code cell. The code is: find_maxval([67, 89, 123, 67, 25, 79]).* [Video description ends]

And on this occasion, the number of iteration required to find the maximum value of 123 is 6.

[Video description begins] *He enters a code in the twenty-fifth code cell. The code is: find_maxval([67, 89, 12, 67, 25, 79, 10, 433, 6, 4, 90, 78]).* [Video description ends]

So when we run this function with an array of 12 integers, this also runs in 12 iterations, showing the fact that the complexity of this function is Big O(n).

[Video description begins] *He executes the code in the twenty-fifth code cell. The output reads: Maximum value of the list = 433 Total number of iterations = 12. The following message is displayed: Number of iterations scales linearly with input size - complexity is O(n).* [Video description ends]

We will now continue taking a look at operations which have complexities of Big O(n), and this includes 1, where we will need to find the factorial of a given number. So this function find_factorial accepts an integer as an input.

[Video description begins] *He enters a code in the twenty-sixth code cell. The code is, code starts: def find_factorial(number:int): fact = 1 num_iterations = 0 if (number < 0 or type(number) != int): print("Invalid number") return for i in range(1, number+1): fact = fact * i num_iterations += 1 print ("The factorial of %d = %d \nTotal number of iterations = %d" %(number, fact, num_iterations)). Code ends.* [Video description ends]

And then, in order to calculate the factorial, it will make use of this fact variable which is initialized to 1, and will also keep track of the number of iterations. This function though, will include a step at first in order to check whether the input happens to be a valid one. So here, we confirm that the number is not less than 0, and that it is indeed an integer. Now this is a constant number of steps that should be executed each time the function is called. So the total number of steps involved when executing this function will not only involve the for loop which is just to come a little later, but also a set number of steps in order to execute this if condition.

So if the number happens to be negative or not an integer, we exit from the function. However, if it is a positive integer, we proceed with the execution, and then calculate the factorial of that number for which we make use of this for loop. Now to calculate this factorial though, we will need to start off with the number 1, which is the initial value of the fact variable. And then we incrementally multiply by the next integer, until we get to the input argument. So at each step in the for loop, we increment num_iterations. And then at the end of the for loop, we will print out the value of the factorial along with the original input, along with the total number of iterations which are required to calculate the factorial.

[Video description begins] *He enters a code in the twenty-seventh code cell. The code is: find_factorial(1).* [Video description ends]

So we will now call this find_factorial function by first calculating the factorial of 1, which is correctly returned as 1. And the number of iterations required to perform this calculation was 1. What about finding the factorial of 8?

[Video description begins] *He enters a code in the twenty-eighth code cell. The code is: find_factorial(8).* [Video description ends]

Well, this has been correctly calculated as 40,320, but the number of iterations was equal to the value of our input. In fact, if we try to find the factorial of the number 1,050, then this output is extremely large value for the factorial, but the total number of iterations required to perform this calculation was equal to the value of our input.

[Video description begins] *He enters a code in the twenty-ninth code cell. The code is: find_factorial(1050).* [Video description ends]

So just as we saw with our check prime functions, the complexity of this operation is Big O (n). And we see this, even though there was a constant set of operations in order to carry out the if statement, which was executed before the for loop. To illustrate this just a little more clearly, we will now include one more function, and this is one which includes two_for_loops within it. One of these runs a constant number of times, specifically 100 times, while the number of iterations of the other for loop depends on the input to this function.

[Video description begins] *He enters a code in the thirtieth code cell. The code is, code starts: def two_for_loops(number): total = 0 num_iterations = 0 for i in range(number): num_iterations += 1 total = total + i for i in range(100): num_iterations += 1 total = total + i print("Total iterations are %d" % num_iterations). Code ends*. [Video description ends]

So this is for a function called two_for_loops, and then once it accepts the argument, we will first initialize two variables called total and num_iterations to 0. So the first for loop is dependent on the argument which is passed to this function. So within the body of the for loop, we increment the value of num_iterations, and we calculate a running total, which at the end of this for loop, will have calculated the sum of all integers from zero until the input argument. We then include another for loop, which will execute a constant number of times, no matter what the input argument. So this will run exactly 100 times, and within this, we will also update the value of num_iterations and the total variable.

So both of these for loops perform exactly the same operations, it's just that the number of times each of them run can vary. If the input argument happens to be n, then the first for loop will run n times, while the second 1 will run 100 times. So we should expect to see a total of n plus 100 iterations each time this function is called. And it is this number of iterations which we print out at the end. All right, so we call two_for_loops with an input argument of 100.

[Video description begins] *He enters a code in the thirty-first code cell. The code is: two_for_loops(100)*. [Video description ends]

And in this case, each of the for loops will have exactly the same number of iterations, which is why we see a total number of iterations of 200 with this invocation. What if our argument is 1,000?

[Video description begins] *He enters a code in the thirty-second code cell. The code is: two_for_loops(1000)*. [Video description ends]

Well, the number of iterations is 1,100. So it is n plus 100 once again. In fact, this will also apply if our argument happens to be 1 million.

[Video description begins] *He enters a code in the thirty-third code cell. The code is: two_for_loops(1000000)*. [Video description ends]

So what is the complexity of such an operation?

[Video description begins] *He executes the code in the thirty-third code cell. The output reads: Total iterations are 1000100*. [Video description ends]

As I had mentioned earlier with the factorial calculation, it is still represented as Big O(n). This is because when we consider the complexity of any task, we assume that the value of n is very, very large, and that any constant values can simply be ignored from the calculation. So the purpose of the Big O notation, is not to provide to you a highly accurate count of the number of operations which will be performed by an algorithm, but to give you a fairly big picture view, of how the number of iterations will vary depending on the input.

O(n*n) Operations

[Video description begins] *Topic title: O(n*n) Operations. The presenter is Kishan Iyer.* [Video description ends]

We now continue with our time complexity Jupyter notebook. But in this video, we will take a look at operations which have a time complexity of Big O (n) square. So the function we will use to demonstrate this is called print_pairs.

[Video description begins] *He opens the Time Complexity file in the Jupyter Notebook window*. [Video description ends]

And this accepts as an argument a list of integers. What it does contain within its body is a nested for loop. And this is one of the most obvious indications of an algorithm which will run in Big O (n) square time. We can now delve within this function to understand exactly how that works.

[Video description begins] *He enters a code in the thirty-fourth code cell. The code is, code starts: def print_pairs(number_list): num_iterations = 0 n = len(number_list) for i in range(n): for j in range(n): print(number_list[i], number_list[j]) num_iterations += 1 print("Total iterations are %d" % num_iterations). Code ends*. [Video description ends]

So this print_pairs function is meant to print all permutations of pairs for numbers in the given list. We initialize num_iterations to zero, and then we set a value of n which is equal to the length of our numbers list. And then we initialize our first for loop, which will iterate through each of the elements within our numbers list. And in order to generate all possible permutations of pairs, we will need to go through one iteration of the entire list for each element within the list, which is why we have this nested for loop. It is within the inner for loop where we print out the given permutation of numbers. And then we will also increment the value of num_iterations. Once both these for loops have finished execution, we will print out the value of num_iterations.

[Video description begins] *He enters a code in the thirty-fifth code cell. The code is: print_pairs([123, 67])*. [Video description ends]

With our function definition complete, we can run the cell, and we invoke print_pairs, and we pass in a list of two integers. As you would expect, the total number of permutations of pairs is four. This is in fact equal to the square of the size of the array, which is why we see that the total number of iterations is also four. What if we called print_pairs using a list which has four elements?

[Video description begins] *He enters a code in the thirty-sixth code cell. The code is: print_pairs([123, 67, 25, 79])*. [Video description ends]

There are 16 different permutations of pairs, and there are total of 16 iterations. So this is an algorithm which has a complexity of Big O (n) square.

[Video description begins] *The following message is displayed: Number of iterations is equal to the square of the input size - complexity is O(n*n)*. [Video description ends]

The reason we get this complexity is because the first for loop ran n times. And for each of those n iterations of the outer for loop, we had n iterations of the inner for loop. And this is a classic case. We will now move onto an example where we once again have a nested for loop. But the number of iterations for each of the for loops here is not dependent on the same value, but on two different values. So here, the aim of tise find_prime function is to identify all of the prime numbers in a given range of values.

[Video description begins] *He enters a code in the thirty-seventh code cell. The code is, code starts: def find_prime(lower, upper): num_iterations = 0 for num in range(lower, upper): for i in range(2, int(num / 2)): num_iterations += 1 if(num % i) == 0: break else: print(num) print("Total iterations are %d" % num_iterations). Code ends*. [Video description ends]

So the arguments to find_prime include this range of values where you specify a lower bound and an upper bound. And for every integer within that range, it will determine whether that number happens to be prime or not. Once again, we initialize num_iterations to zero. And then we move on to our outer for loop where we iterate through all of the values within the given range. So if you consider that the range of values here represents m different elements. Here I'm referring to m for Maria. And within this inner for loop, we use the same logic as we did previously in our second version of the check_prime function, where we iterate through all of the integers between two and half of a given number in order to check whether it is a prime number.

The number of iterations in this for loop will depend on the value of the number whereas the number of iterations in the outer for loop depends on the range of the given numbers. So within our inner for loop, we first increment num_iterations. And then we perform a divisibility check. If the number manages to get divided by another number without leaving a remainder, it means that this number is not a prime number. In which case we break out of the inner for loop and then move ahead to the next number within the range. The else statement included here is the else for the inner for loop, which the code will get to at the end of the inner for loop as long as no break statement has been encountered. In other words, this else statement is executed if the number being evaluated in the inner for loop happens to be a prime number.

So in this case, we will print out the value of this prime number. And finally, once the outer for loop has also done executing, we will print out the total number of iterations.

[Video description begins] *He enters a code in the thirty-eighth code cell. The code is: find_prime(10, 20).* [Video description ends]

All right, so we'll now run this cell, and then call the find_prime function in order to identify all of the prime numbers between 10 and 20. The size of this range is 10, which is why we know that the outer for loop will execute ten times. As for the inner for loop, though, this will vary depending on the number. For any even number, there'll just be a single iteration. But for the largest prime number in this range, which is 19, a total of 7 iterations will be required to identify it as a prime.

So when we run this function, we see that in order to identify the four prime numbers of 11, 13, 17, and 19, there were a total of 27 iterations which were required. So if the size of our range is m for Maria, then the complexity of the outer for loop is Big O (m). However the complexity of the inner for loop we can say is Big O (n) for Nancy. In which case, we can say that the complexity of this entire function is Big O (n) times m.

[Video description begins] *He enters a code in the thirty-ninth code cell. The code is: find_prime(10, 2000).* [Video description ends]

This also applies if you were to increase the range of values. And in this particular function call, the value of the prime numbers to identify is also much larger. So when we run this, the output does not really fit into a screen. So Jupyter notebooks prints out this output in an area which we can scroll over. So I'm just going to scroll all the way to the bottom. And along the way, we can see all of the prime numbers between 10 and 2,000. But to get the total number of iterations, we check out the number at the bottom, which in this case is 143,122. Now to better illustrate any algorithm with a complexity of n times m, we will now use a simpler function. This is called print_combinations. And this is similar to the function which we saw previously, which had a complexity of Big O (n) square, in that this function also includes a nested for loop.

[Video description begins] *He enters a code in the fortieth code cell. The code is, code starts: def print_combinations(n, m): num_iterations = 0 for i in range(n): for j in range(m): print(i, j) num_iterations += 1 print("Total iterations are %d" % num_iterations). Code ends.* [Video description ends]

However, there are two arguments which are specified, which are n and m, and the number of iterations for each of the for loops depends on these values. We initialize our num_iterations variable. And then there is an outer for loop which will iterate over all of the values between zero and n for Nancy. The inner for loop will iterate over the values between zero and m for Maria. Each possible permutation is printed out within this inner for loop, and num_iterations is also incremented. So we should expect that the num_iterations will be n times m. And we can confirm that by printing out the num_iterations at the end. So we can now run the cell and then call our new print_combinations function.

[Video description begins] *He enters a code in the forty-first code cell. The code is: print_combinations(10, 12).* [Video description ends]

And we should expect a total of 120 different permutations to be printed out. So we can see each of these permutations here. But once we scroll all the way to the bottom, we get the confirmation that the number of iterations is indeed 120. So this is a case where the complexity of an algorithm is not dependent on a single variable, but on two different variables. And with that, we can conclude this demo on time complexity, where we have now seen various algorithms which vary in complexity from Big O (1) to Big O (n), Big O (n) square, and Big O (n) times m.

Python's Built-in Queue

[Video description begins] *Topic title: Python's Built-in Queue. The presenter is Kishan Iyer.* [Video description ends]

We now move on to the second demo in this course where we will be demonstrating the queue data structure available in Python.

[Video description begins] *He opens the Queue file in the Jupyter Notebook window.* [Video description ends]

So I have created a new Jupyter notebook for this, which I have named Queue.

[Video description begins] *He enters a code in the first code cell. The code is: from queue import Queue.* [Video description ends]

To start off with, we can import the queue class from the queue module. In most high-level programming languages, you will never need to implement a queue. It is such a fundamental data structure that every language will have its own version of the queue data structure. All we will need to do is to make it available within our source code which is what we are doing right here.

[Video description begins] *He enters a code in the second code cell. The code is: olympics = Queue(5) olympics.* [Video description ends]

So once we have the queue class, we will now initialize a new queue. And in Python, you can call the queue class constructor and the argument which we pass to it happens to be the length of the queue. So here we create a queue called Olympics and this has a capacity of 5 elements.

[Video description begins] *He executes the code in the second code cell. The output reads: <queue.Queue at 0x10fefb7f0>.* [Video description ends]

And when we view this queue object, we see that this is of type queue.

[Video description begins] *He enters a code in the third code cell. The code is, code starts: olympics.put('United States(USA)') olympics.put('Great Britain(GBR)'). Code ends.* [Video description ends]

Now, in order to load this queue with elements, we will need to make use of its put method. For a Python queue object, this is the equivalent of an enqueue method where a new element gets added to the end of the queue. So the first element to get added to the queue is a string called United States. Because we will be using this Olympics queue in order to represent the five different countries which tallied the most number of medals in the 2016 Olympics. So we add the string United States, followed by the string Great Britain. So once we run these operations, we can confirm whether elements did indeed get loaded onto the queue.

[Video description begins] *He enters a code in the fourth code cell. The code is: olympics.empty().* [Video description ends]

And for that, we can use the built in method of the queue object, which is empty. So this is equivalent to the is empty function which we discussed during the theoretical section of this learning path. And it will return True if the queue is currently empty. And when you run the cell we get a value of False since the queue does indeed have some elements within it.

[Video description begins] *He enters a code in the fifth code cell. The code is: olympics.full()*. [Video description ends]

To check whether the queue is full, there is a method called full for the queue object. And this also returns a value of False because the queue still has three empty slots.

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

Now to check how much of the queue is currently occupied, we need to check the size of the queue for which there is a function called qsize.

[Video description begins] *The following message is displayed: empty(), full() and qsize() are O(1) operations*. [Video description ends]

You can expect that operation such as empty, full, and qsize will be Big O(1) operations. That is, operations which run in constant time and don't depend on the size of the queue or the number of elements which are loaded within it. This can be ensured by implementing variables which keep track of the size as well as the capacity of the queue. So when we run Olympics.qsize, and this correctly returns a value of 2.

[Video description begins] *He enters a code in the seventh code cell. The code is, code starts: olympics.put('China(CHN)') olympics.put('Russia(RUS)') olympics.put('Germany(GER)'). Code ends*. [Video description ends]

All right, it is now time for us to start filling up the queue. So we now loaded three additional values to represent the countries which are number three, number four, and number five in the total medal count in these Olympics.

[Video description begins] *He enters a code in the eighth code cell. The code is: olympics.empty()*. [Video description ends]

And once that is done, we once again perform a check to see whether the queue is empty, and of course, it is not.

[Video description begins] *He executes the code in the eighth code cell. The output reads: False*. [Video description ends]

But now that we have loaded five different elements into it, is the queue full?

[Video description begins] *He enters a code in the ninth code cell. The code is: olympics.full()*. [Video description ends]

And the answer which we get is yes.

[Video description begins] *He executes the code in the ninth code cell. The output reads: True*. [Video description ends]

So we can confirm that the size of the queue is five by using the qsize function.

[Video description begins] *He enters a code in the tenth code cell. The code is: olympics.qsize()*. [Video description ends]

And yes, the number of elements in the queue is equal to the capacity of the queue.

[Video description begins] *He executes the code in the tenth code cell. The output reads: 5*. [Video description ends]

All right, so now that we have a full queue, let us say we want to use the same queue in order to process additional countries. So to do that, we will need to get elements out of the queue and one way to do that is to dequeue the first element within the queue.

[Video description begins] *He enters a code in the eleventh code cell. The code is: olympics.get()*. [Video description ends]

So just as put happens to be the equivalent of the enqueue function for a Python queue object, it is the get function, which is the equivalent of the dequeue function. A queue data structure follows the first in, first out policy. And the first element which we loaded into our queue represented the United States. So is this the element which gets out of the queue when we run get?

[Video description begins] *He executes the code in the eleventh code cell. The output reads: 'United States(USA)'*. [Video description ends]

The answer is yes, so we should expect that United States is now no longer in the queue. We can now get some confirmation of that by running the queue size method once more.

[Video description begins] *He enters a code in the twelfth code cell. The code is: olympics.qsize()*. [Video description ends]

And this shows that the queue now contains one fewer element, that is, four elements compared to the previous five.

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

We can now check whether the queue is full and that is no longer the case, so we clearly have capacity for additional elements.

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

However, we will now continue getting elements out of the queue.

[Video description begins] *He executes the code in the thirteenth code cell. The output reads: False*. [Video description ends]

So we will run get once more.

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

And we should expect that out of the existing elements in the queue, it is the one which was added first which gets dequeued when we run this function. And in fact, it is indeed Great Britain which gets dequeued.

[Video description begins] *He executes the code in the fourteenth code cell. The output reads: 'Great Britain(GBR)'*. [Video description ends]

All right, we run Olympics.get once more.

[Video description begins] *He enters a code in the fifteenth code cell. The code is: olympics.get()*. [Video description ends]

And it is China, which is now off the queue.

[Video description begins] *He executes the code in the fifteenth code cell. The output reads: 'China(CHN)'*. [Video description ends]

When we run it for a fourth time, we should expect that Russia gets out.

[Video description begins] *He enters a code in the sixteenth code cell. The code is: olympics.get()*. [Video description ends]

And yes, that is what happens.

[Video description begins] *He executes the code in the sixteenth code cell. The output reads: 'Russia(RUS)'*. [Video description ends]

And there is still one more element in the queue. So we can run get for the fifth time.

[Video description begins] *He enters a code in the seventeenth code cell. The code is: olympics.get()*. [Video description ends]

And now, Germany is also out of the queue.

[Video description begins] *He executes the code in the seventeenth code cell. The output reads: 'Germany(GER)'*. [Video description ends]

So if the queue empty now as we expect, for that, we run the empty method.

[Video description begins] *He enters a code in the eighteenth code cell. The code is: olympics.empty()*. [Video description ends]

And this returns a value of True, confirming that the queue is indeed empty. With that, we have now seen how we can make use of Python's native queue object in order to create a queue data structure. It includes all of the functions which we require in order to work with a queue. And if you're building your own Python application in which you require a queue, then you can simply make use of the queue object which is already available. However, there may be many programming interviews where you may be asked to implement a queue in Python using a list structure. And that is what we will cover in the next video.

Defining a Custom Queue

[Video description begins] *Topic title: Defining a Custom Queue. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we saw how we can make use of the built in queue data structure which is available in Python. In order to work with a queue, we will now write a class for our own queue in which we will store the elements of a queue in a Python list.

[Video description begins] *He opens the Queue file in the Jupyter Notebook window.* [Video description ends]

So to begin, we will create a new class called MyQueue.

[Video description begins] *He enters a code in the nineteenth code cell. The code is, code starts: class MyQueue: def _init_(self): self.items = []. Code ends*. [Video description ends]

And then within that, we define the init function, which effectively serves as the constructor for this queue. And this does not really take in any arguments. What it does though is initialize a list called items which is set to an empty list. So in this particular case, we're not really setting a capacity for the queue, and we can go on adding elements to this queue. So now that the init function has been defined, we move along to the next function, and this is where we implement the is_empty operation.

[Video description begins] *In the nineteenth code cell, after the line of code: self.items = [], he adds a code. The code is, code starts: def is_empty(self): return len(self.items) == 0. Code ends.* [Video description ends]

There are no arguments for this function and it returns a Boolean value, which represents whether the queue is empty or not. To do that, we check the number of items available within the items list. And if it is zero, we return True. Moving along now to the enqueue function.

[Video description begins] *In the nineteenth code cell, after the line of code: return len(self.items) == 0, he adds a code. The code is, code starts: def enqueue(self, item): self.items.append(item). Code ends.* [Video description ends]

And this is the one which is used in order to add an element or an item to our queue. The argument which this accepts is the item which needs to be added to the queue. And then we make use of the append function of a Python list in order to add the item to our list of items. Do make note of the fact that by using the append function, we are adding this item to the end of the queue, which is exactly what the enqueue operation must do. The next function to define is the dequeue function, and this is one where the item at the front of the queue is taken off the queue.

[Video description begins] *In the nineteenth code cell, after the line of code: self.items.append(item), he adds a code. The code is, code starts: def dequeue(self): return self.items.pop(0). Code ends.* [Video description ends]

The way to do that when using a list data structure is to call the pop method. And the argument to the pop method is the index of the element which needs to be removed from the list. In our case, we wish to remove the element at the front of the queue which is at index zero in the items list. All right, so the enqueue and dequeue functions are taken care of. Another function which we will implement now returns the number of elements which are currently in the queue and this is the function called size.

[Video description begins] *In the nineteenth code cell, after the line of code: return self.items.pop(0), he adds a code. The code is, code starts: def size(self): return len(self.items). Code ends.* [Video description ends]

Since the underlying data structure in our queue implementation is a list, all we need to do is to calculate the length of the items list and return this. Moving on now to the peek function.

[Video description begins] *In the nineteenth code cell, after the line of code: return len(self.items), he adds a code. The code is, code starts: def peek(self): if self.is_empty(): raise Exception("Nothing to peek") return self.items[0]. Code ends.* [Video description ends]

And this is the one which will return to us the value of the element at the front of the list. The peek function does not really modify the queue in any manner and it does not take in any arguments either. However, when we implement the peak operation, we first need to perform a check to see whether the queue is empty. For this, we make use of the is_empty function which we have already defined. And it will throw an exception in case of an empty queue. In which case we catch the exception and then we print out the message nothing to peek.

All right, so if the exception is not thrown, which means that you have some elements, what we do is to return the value of the first element. This is available in items at index zero. With all of these functions defined, we have now returned the code for our own custom queue. Let us go ahead and run the cell. And then we will create our first custom queue.

[Video description begins] *He enters a code in the twentieth code cell. The code is: olympics = MyQueue() olympics.* [Video description ends]

So once again, this is a variable called Olympics, and it is initialized to MyQueue. The constructor to MyQueue doesn't really take in any arguments and will just initialize an empty list. And when we check out our Olympics variable, we are informed that this is of type MyQueue.

[Video description begins] *He executes the code in the twentieth code cell. The output reads: <_main_.MyQueue at 0x10ff2ba58>.* [Video description ends]

All right, so our initialization has been successful. Let us go ahead and add a few elements to it.

[Video description begins] *He enters a code in the twenty-first code cell. The code is, code starts: olympics.enqueue('United States(USA)') olympics.enqueue('Great Britain(GBR)'). Code ends.* [Video description ends]

We use our enqueue function of the Olympics object, and we will add the string United States and also Great Britain. So we're loading two elements into our queue and beyond that, check that the two elements have been added, we access olympics.items.

[Video description begins] *He enters a code in the twenty-second code cell. The code is: olympics.items.* [Video description ends]

The output confirms that the two strings have indeed been added to our custom queue.

[Video description begins] *He executes the code in the twenty-second code cell. The output reads: ['United States(USA)', 'Great Britain(GBR)'].* [Video description ends]

So what is the element at the head of the queue?

[Video description begins] *He enters a code in the twenty-third code cell. The code is: olympics.peek().* [Video description ends]

For that, we can take a peek and this returns the value of United States. All right, we can now continue testing the other functions which we defined within our custom queue. So we invoke the size method, and this returns to us a value of 2 representing the two elements which we added.

[Video description begins] *He enters a code in the twenty-fourth code cell. The code is: olympics.size().* [Video description ends]

Things are looking good so far. So we will continue playing around with our custom queue by enqueuing three additional elements.

[Video description begins] *He enters a code in the twenty-fifth code cell. The code is, code starts: olympics.enqueue('China(CHN)') olympics.enqueue('Russia(RUS)') olympics.enqueue('Germany(GER)') olympics.size(). Code ends.* [Video description ends]

So these are the strings, China, Russia, and Germany. And once all of them have been enqueued, we will also check the size of our Olympics queue. And this correctly returns a value of five. When we peek into our queue again, it's the United States which is still at the front of the queue.

[Video description begins] *He enters a code in the twenty-sixth code cell. The code is: olympics.peek().* [Video description ends]

So what happens if we invoke the dequeue function?

[Video description begins] *He enters a code in the twenty-seventh code cell. The code is: olympics.dequeue().* [Video description ends]

We are returned the value of United States, so it looks like that has been popped from the queue. And now when we execute the peek function, it is the next element which was introduced in the queue, specifically Great Britain, which is now at the head of the queue.

[Video description begins] *He enters a code in the twenty-eighth code cell. The code is: olympics.peek().* [Video description ends]

Just to emphasize that the peek function is a read-only operation and does not modify the queue in any way, we will invoke this one more time, and this returns the exact same value of Great Britain.

[Video description begins] *He enters a code in the twenty-ninth code cell. The code is: olympics.peek().* [Video description ends]

We execute dequeue for a second time.

[Video description begins] *He enters a code in the thirtieth code cell. The code is: olympics.dequeue().* [Video description ends]

Now Great Britain has been popped out of the queue.

[Video description begins] *He enters a code in the thirty-first code cell. The code is: olympics.dequeue().* [Video description ends]

Running dequeue for the third time, this gives us a value of China and we execute dequeue for the fourth time.

[Video description begins] *He enters a code in the thirty-second code cell. The code is: olympics.dequeue().* [Video description ends]

So now Russia is also out of the queue. At this point, we should expect that it is only Germany which is still in our queue. So when we take a peek, it is Germany which has returned.

[Video description begins] *He enters a code in the thirty-third code cell. The code is: olympics.peek().* [Video description ends]

Just to confirm that there is just one element left within our queue, we invoke the size function once more, and this returns the value of one.

[Video description begins] *He enters a code in the thirty-fourth code cell. The code is: olympics.size().* [Video description ends]

So if you were to run the dequeue operation once more from our Olympics object, this should return to us the value of Germany.

[Video description begins] *He enters a code in the thirty-fifth code cell. The code is: olympics.dequeue().* [Video description ends]

Now what if you were to execute dequeue one more time?

[Video description begins] *He enters a code in the thirty-sixth code cell. The code is: olympics.dequeue().* [Video description ends]

So within the definition of our dequeue function, we do not really account for the error case where we check for an empty list. As a result, when we run the cell, this will raise an exception, specifically an index error is thrown because we are trying to pop the first element from an empty list.

[Video description begins] *He executes the code in the thirty-sixth code cell. An error is raised.* [Video description ends]

When implementing a queue data structure on your own, you must be careful in accounting for all of these boundary conditions. You must not allow dequeuing from an empty queue, and in case your queue has a maximum capacity, you should not allow the enqueue operations when the queue is full.

Use a Python List as a Stack

[Video description begins] *Topic title: Use a Python List as a Stack. The presenter is Kishan Iyer.* [Video description ends]

Previously, we took a look at how we can use queues in Python. We explored two different ways in which to do that. One of those was using the built-in queue class available in Python. And the other approach was to define our own custom queue class in which we store the elements of a queue in a Python list. In this demo, we will see how we can implement stacks in Python.

[Video description begins] *He opens the Stack file in the Jupyter Notebook window.* [Video description ends]

So here, I have created a new notebook, which I have named Stack. And to start off with, we will initialize an empty list and call this stack.

[Video description begins] *He enters a code in the first code cell. The code is: stack = [].* [Video description ends]

You add it to the top of the stack.

[Video description begins] *He enters a code in the second code cell. The code is, code starts: stack.append('United States(USA)') stack.append('Great Britain(GBR)') stack.append('China(CHN)'). Code ends.* [Video description ends]

This is precisely what is accomplished when we execute the append function for a list in Python. So here, we will run the append function and add three items to our stack.

[Video description begins] *He enters a code in the third code cell. The code is: stack.* [Video description ends]

Following that, when we take a look at the current contents of our stack, we see that it is the last item which is added to the stack, which is at the top of the stack, specifically the string China.

[Video description begins] *He executes the code in the third code cell. The output reads: ['United States(USA)', 'Great Britain(GBR)', 'China(CHN)'].* [Video description ends]

It is the United States which was the first to be added to our stack, which means that if you were to execute the pop function, we should expect that it is China which will be popped out of the stack first.

[Video description begins] *He enters a code in the fourth code cell. The code is: stack.pop().* [Video description ends]

And that the United States will be the last one to be popped out. So what happens if we were to execute the pop function for a Python list?

[Video description begins] *He executes the code in the fourth code cell. The output reads: 'China(CHN)'.* [Video description ends]

Well, just as you would expect in any stack data structure, it is the last element to be added, which is the first one to be removed. That is, a stack data structure, as well as a Python list, follows the last in first out policy. You may recollect that in the earlier demo for a queue, we were able to pop elements from the front of a list. This is because a Python list is a little more flexible than a regular stack, but you can see that by default it is built to behave as a stack. When calling the append function, you add to the end of the list or add to the top of the stack, if you will.

[Video description begins] *He enters a code in the fifth code cell. The code is: stack.* [Video description ends]

And when you run the pop function without specifying an argument, by default, it is the last element to be added, which is the one that gets removed. That is, last in first out. So now that we have popped one element of the stack, let us take a look at its contents. As you would expect, it still contains two elements within it.

[Video description begins] *He executes the code in the fifth code cell. The output reads: [‘United States(USA)’,‘Great Britain(GBR)’].* [Video description ends]

We can now re-add China back to the stack.

[Video description begins] *He enters a code in the sixth code cell. The code is: stack.append('China(CHN)').* [Video description ends]

So we run append once more.

[Video description begins] *He enters a code in the seventh code cell. The code is: stack.* [Video description ends]

And when we examine the contents of the stack, we see that it gets added to the back.

[Video description begins] *He executes the code in the seventh code cell. The output reads: [‘United States(USA)’, ‘Great Britain(GBR)’, 'China(CHN)'].* [Video description ends]

So the end of the Python list and the top of a stack are pretty much the same. All right, we'll now move ahead and add a few additional elements to the stack.

[Video description begins] *He enters a code in the eighth code cell. The code is, code starts: stack.append(‘Russia(RUS)’) stack.append(‘Germany(GER)’) stack.append(‘Japan(JPN)’) stack.append(‘France(FRA)’) stack.append(‘South Korea(KOR)’) stack.append(‘Italy(ITA)’) stack.append(‘Australia(AUS)’). Code ends.* [Video description ends]

Specifically, we will add seven elements so that our stack will now contain a total of 10 different values.

[Video description begins] *He enters a code in the ninth code cell. The code is: stack.pop().* [Video description ends]

Now if you were to execute the pop function, it is the last element which was added, that is Australia, which gets removed.

[Video description begins] *He executes the code in the ninth code cell. The output reads: ‘Australia(AUS)’.* [Video description ends]

And if you execute pop once more, it is Italy which gets popped off. So if you would like to make use of a basic stack data structure in Python, all you need to do is to create a Python list. However, this may be insufficient for many operations on a stack. For example, if you would like to catch an exception in case someone tries to pop an element from an empty stack, you would need to implement such behavior using your own custom stack. This is now precisely what we will do, where we will create a new class called stack. This is something we will implement in the next video.

Defining a Custom Stack

[Video description begins] *Topic title: Defining a Custom Stack. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we saw that we can use a Python list effectively as a stack. However, it may be desirable to extend the behavior of a regular Python list in order to implement additional operations within a stack. To do that, we will now define our own custom class called stack.

[Video description begins] *He opens the Stack file in the Jupyter Notebook window.* [Video description ends]

Within the constructor for this stack class, we will initialize an empty list called stack.

[Video description begins] *He enters a code in the eleventh code cell. The code is, code starts: class Stack: def _init_(self): self.stack = []. Code ends.* [Video description ends]

And then following that, we will go ahead and define each of the functions for our own custom stack.

[Video description begins] *In the eleventh code cell, after the line of code: self.stack = [], he adds a code. The code is, code starts: def push(self, data): self.stack.append(data). Code ends.* [Video description ends]

The push function will be used in order to add an element to the top of the stack which accepts an argument which is the data to be added to the stack. And then internally, we make use of the append function of a list in order to add the data which is input to the top of the stack. That is to the end of the list.

[Video description begins] *In the eleventh code cell, after the line of code: self.stack.append(data), he adds a code. The code is, code starts: def pop(self): if self.is_empty(): raise Exception("Nothing to pop") return self.stack.pop(len(self.stack) - 1). Code ends.* [Video description ends]

Now when we define our pop function, this does not accept any argument, and then removes the element from the top of the stack. However, we do wish to implement the behavior where it does not throw an error if we try to run the pop function on an empty stack. So to check for an empty stack, we will call the is_empty function here. This is a function within our stack class which we will define in just a little bit. This will return a True or False value depending on whether the stack is empty or not. And if it is empty, we will raise our own custom exception where we print out that there is nothing in the stack to pop.

However, if the stack does contain elements, we will return the item at the top of the stack. And to do that, we invoke the pop function of the Python list. Here we explicitly specify the index of the last element in our stack. So once the pop function has been defined, we have redefined the function which we already saw in a Python list. We will now implement an additional function. And this is the one to peek at the top of the stack for the function called peek does not accept any argument.

[Video description begins] *In the eleventh code cell, after the line of code: return self.stack.pop(len(self.stack) - 1), he adds a code. The code is, code starts: def peek(self) if self.is_empty(): raise Exception("Nothing to peek") return self.stack[len(self.stack) - 1]. Code ends.* [Video description ends]

And this also performs a check to see whether the stack is empty. And if not, just like with the pop function, it also accesses the element at the end of the list or the top of the stack. But unlike the pop operation, which we use in the pop function, here, we simply return the value at the top of the stack. This is in keeping with the behavior of a peek operation, which is read only.

[Video description begins] *In the eleventh code cell, after the line of code: return self.stack[len(self.stack) - 1], he adds a code. The code is, code starts: def is_empty(self): return len(self.stack) == 0. Code ends.* [Video description ends]

We will now come to the definition of the is_empty function, which we have already called both in the pop as well as the peek functions. Here we return a value of True if the length of the internal stack list is 0.

[Video description begins] *In the eleventh code cell, after the line of code: return len(self.stack) == 0, he adds a code. The code is, code starts: def size(self): return len(self.stack). Code ends.* [Video description ends]

Following the is_empty function, we also implement a function called size which returns the number of elements which are currently in our stack. We can get this value by performing a length operation on the stack list. This concludes our definition for all of the operations in our custom stack. And you will observe that the complexity for each of these operations will be Big O(1) because they will execute in constant time, no matter what the size of our stack, since you're never iterating over the contents of the stack.

[Video description begins] *He enters a code in the twelfth code cell. The code is: olympics = Stack() olympics.* [Video description ends]

So we move along now and then redefine our Olympics variable, and this time it's going to be of type stack.

[Video description begins] *He executes the code in the twelfth code cell. The output reads: <_main_.Stack at 0x1039086a0>.* [Video description ends]

We can in fact confirm that by running the cell which shows that this is indeed of type stack, and moving along we will append four different values to our stack.

[Video description begins] *He enters a code in the thirteenth code cell. The code is, code starts: olympics.push('United States(USA)') olympics.push('Great Britain(GBR)') olympics.push('China(CHN)') olympics.push('Russia(RUS)'). Code ends.* [Video description ends]

To do that, we make use of the push function which we defined. So we would push the values United States, Great Britain, China, and Russia.

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

Following that, we invoke the size function to check number of elements in the stack. And by returning a value of 4, we can be confident that our push function, as well as the size function, have been implemented correctly. We will now invoke the push function a couple more times.

[Video description begins] *He enters a code in the fifteenth code cell. The code is, code starts: olympics.push('Germany(GER)') olympics.push('Japan(JPN)'). Code ends.* [Video description ends]

So we add Germany and Japan to our stack as well, and we shall expect that when we invoke the pop function, it is the last element to go in, specifically Japan, which is the first element to pop out of the stack.

[Video description begins] *He enters a code in the sixteenth code cell. The code is: olympics.pop().* [Video description ends]

And yes, we do have that confirmation right here.

[Video description begins] *He executes the code in the sixteenth code cell. The output reads: 'Japan(JPN)'.* [Video description ends]

There are still a few other functions in our custom stack which we are yet to test out. So we will now invoke is_empty, which unsurprisingly returns a value of False.

[Video description begins] *He enters a code in the seventeenth code cell. The code is: olympics.is_empty().* [Video description ends]

We can now test out the peek function, which will return the value at the top of the stack.

[Video description begins] *He enters a code in the eighteenth code cell. The code is: olympics.peek().* [Video description ends]

And this returns the string Germany. Since the peek at the top of the stack showed us Germany, we should expect that running the pop operation will also return Germany.

[Video description begins] *He enters a code in the nineteenth code cell. The code is: olympics.pop().* [Video description ends]

And that is precisely what occurs. If you were to run pop one more time, the previous element to be added to the stack gets popped out, which is Russia. And we will continue executing the pop function until all the elements come out, in which case we expect that our custom exception will be thrown. So now China comes off the stack, followed by Great Britain. And when we execute one more time, it is the United States, which was the first in which is the last out of the stack. So what happens when we execute pop one more time on an empty stack?

It is our custom exception, which prints out the message nothing to pop which is thrown. We can in fact confirm that the stack is empty by invoking is_empty once more. And on this occasion, it returns a value of True. So by using a custom stack, we can extend the behavior of a regular Python list in order to work as a stack data structure which includes a number of additional features over and above what a regular Python list has to offer.

Linked Lists: Defining Insert Operations

[Video description begins] *Topic title: Linked Lists: Defining Insert Operations. The presenter is Kishan Iyer.* [Video description ends]

Previously in this course, we have covered how you can implement your own custom queues as well as custom stacks on top of a Python list. We will now see how we can implement a linked list in Python. The implementation of your own custom data structures are common questions in many programming interviews. But even if you are not studying for any interview and you only wish to understand data structures in general, it is still a very good exercise to implement these using standard Python objects.

[Video description begins] *He opens the LinkedList file in the Jupyter Notebook window.* [Video description ends]

For this demo, I have created another Jupyter notebook which I have named LinkedList. And to start off with, we will define what constitutes a node in our linked list.

[Video description begins] *He enters a code in the first code cell. The code is, code starts: class Node: def _init_(self, dataval=None, nextval=None): self.dataval = dataval self.nextval = nextval def _repr_(self): return repr(self.dataval). Code ends.* [Video description ends]

To do this, we will create a new class called Node and this is of course is a node in a singly-linked list. Just to clarify, a singly-linked list is one which contains a pointer to only the next node within the linked list. This is as opposed to a doubly-linked list where there are pointers to both the next as well as the previous nodes. So we move along with the constructor for this node class. So when creating a new node object, we will need the values for the two fields which make up a node. That is the data for the node as well as the pointer to the next node in the linked list.

These will be supplied as the arguments dataval and nextval. And we will set the default value for each of these to None in order to represent an empty node. However, we will store the dataval and the nextval within this node object. There is now one more function we will implement right here, and this is to print out the contents of the node. So this is the _repr_ function, which stand for the representation of the object. Here, we will simply return the contents of the dataval field. So with that node definition complete, it is now time for us to move on to defining our custom linked list.

[Video description begins] *He enters a code in the second code cell. The code is, code starts: class LinkedList: def _init_(self): self.head = None. Code ends.* [Video description ends]

Any implementation of a linked list needs to include a variable which points to the head node of the linked list. This is what we defined within the init function of our linked list where we initialized the head to None. We will be updating this later on when we add elements to the linked list. So when we initialize a new linked list object, it will contain zero nodes. All it will have is a head variable which has been initialized to null. Moving along now to the printable representation of our custom linked list.

[Video description begins] *In the second code cell, after the line of code: self.head = None, he adds a code. The code is, code starts: def _repr_(self): nodes = [] curr = self.head while curr: nodes.append(repr(curr)) curr = curr.nextval return '[' + '->' .join(nodes) + ']'. Code ends.* [Video description ends]

So to represent our linked list as a string, we will first initialize an empty list called nodes. This is one we will use in order to store the values of each of the nodes in our linked list while we iterate over those nodes a little later in this function. So to help with the iteration, we will use this variable called curr which will point to the current node we are iterating over. This is initialized to the head of our linked list. Following that, we have a while loop, where we will iterate over each of the nodes, and the curr variable points to the current node we are on. At each iteration, we will load the string representation of each node onto our nodes list. To do this, we invoke the repr function on the current node.

And if you look at the definition of that function in our node class, you will observe that it represents the data contents of the node. So this data will be appended to our nodes list. Following that, we point the curr variable to the next node in the linked list by updating it to curr.nextval. The nextval field of a node object, of course, points to the next node in the linked list. When we are at the last node in the linked list, the value of curr.nextval will be null, in which case this while loop will terminate. And at the very end, we will simply return the contents of our nodes list separated by an arrow sign. Also note that the string representation of a linked list will begin and also end with square brackets.

Another thing to keep in mind is that since this particular operation involves iterating over each of the nodes in the linked list, the time complexity will be Big O (n). Also note that we are initializing this nodes list, which will contain as many elements as there are nodes in our linked list. So the extra space required for this operation is also Big O (n). So with that done, we will go ahead and define a function to prepend an element to the linked list.

[Video description begins] *In the second code cell, after the line of code: return '[' + '->' .join(nodes) + ']', he adds a code. The code is, code starts: def prepend(self, dataval): self.head = Node(dataval=dataval, nextval=self.head). Code ends.* [Video description ends]

This is one to add an element at the head of the linked list. When adding an element to the linked list, we will need to supply the data contents of the node which is being added. And this is the argument to the prepend function. And then following that, we will first create a new node object. And we will initialize that with the dataval which is being passed. And we will set the next pointer for this particular node to what is currently the head of the linked list. We then update the head of the linked list through this new node which has been created.

You can see that no matter what the size of the linked list, it is exactly the same set of operation which will be performed each time we invoke the prepend function, which is why the complexity of this operation is Big O (1). We move along now to the append function, which involves adding a new element to the end of the linked list.

[Video description begins] *In the second code cell, after the line of code: self.head = Node(dataval=dataval, nextval=self.head), he adds a code. The code is, code starts: def append(self, dataval): if not self.head: self.head = Node(dataval=dataval) return curr = self.head while curr.nextval: curr = curr.nextval curr.nextval = Node(dataval=dataval). Code ends.* [Video description ends]

Just like with the prepend function, we do need the value for the node which is being added to the list, which is the argument to this function. But then, before we append a new node, we will first perform a check. We will see whether the value of the head for the linked list is null, which would mean that the linked list is empty. And if that is the case, we will create a new node by only specifying a value for its data field. We don't need to supply a value for its nextval, because by default, that will be set to null as is required when we are inserting the first node into the linked list. Once we have created this first node for the linked list, we will also update the head variable to point to this new node. We can then exit from the function if this is the case, for which we invoke return.

However, if the new node being appended is not the first node of the linked list we will need to iterate over each of the existing nodes, and then add the new node at the end. For iteration, we will once again make use of a variable called curr, which will point to the node which we are currently iterating over. And then we will make use of a while loop again where we will continue iterating over the list until we hit the last node for which the value of nextval will be null. Once we add the last node of the linked list, we will initialize our new node. Once again, this is done with the data since the nextval will be null for it.

And we update the nextval field for what was previously the last node of the linked list to point to this newly appended node. So our append function definition is now complete, and there is one more insert function which we need to account for. This is one where we insert an element after a particular node. We will walk through the implementation of this function in the next video.

Linked Lists: Search, Delete, and Reverse Operations

[Video description begins] *Topic title: Linked Lists: Search, Delete, and Reverse Operations. The presenter is Kishan Iyer.* [Video description ends]

We will now implement an add after function for our bespoke linked list where we will insert a new node within the linked list after a specific node containing a given value.

[Video description begins] *He opens the LinkedList file in the Jupyter Notebook window.* [Video description ends]

We could say that this is a function to insert an element in the middle of a linked list.

[Video description begins] *He enters a code in the second code cell. The code is, code starts: def add_after(self, middle_dataval, dataval): if middle_dataval is None: print("Data to insert after not specified") return curr = self.head while curr and curr.dataval != middle_dataval: curr = curr.nextval new_node = Node(dataval = dataval) new_node.nextval = curr.nextval curr.nextval = new_node. Code ends.* [Video description ends]

So the arguments to this function include the value of the data of the node after which this new node will be inserted. And this is the middle_dataval, and then also that value of the new node being inserted. This is the argument dataval. We first perform a check to ensure that there is a value which has been specified for middle_dataval. And if not, we will print out the message that the data to insert after has not been given. And if this checks out, you will need to iterate over the linked list until we find the node which contains data, which matches middle_dataval. To make this iteration happen, we once again create a variable called curr to point to the current node we're iterating over. And this of course is initialized to the head. And then we make use of a while loop for the iteration.

And we will continue with this iteration until the dataval field of the current node is equal to middle_dataval. We also terminate the while loop if the value of the current node is null, which of course will occur if we reached the end of the linked list and don't find the data. Once we get out of the while loop, the variable curr will point to either the element whose data matches middle_dataval. Or it will point to the last element in the linked list. So even if we don't find a node whose data matches middle_dataval, we will simply append a new_node at the end of the link list. To do that, we initialize a new_node object using dataval. And then following that, we update the value of nextval for the newly created node to curr.nextval. If the curr pointer happens to point to a node whose data matches the input argument, then the new node will have been inserted in the middle of the linked list.

And if not, it will simply become the new last node of the linked list, whose nextval is null. Whichever node the current pointer was pointing to at the end of the while loop though, will have its next val field updated to point to this newly created node. So what about the time complexity of both the append and the add after functions here? For the append function, we will be iterating over the entire list every single time. And for add after, the same thing will happen in the worst case. So the time complexity is Big O (n). We now move along to the next function for our custom linked list, and this is the find function.

[Video description begins] *In the second code cell, after the line of code: curr.nextval = new_node, he adds a code. The code is, code starts: def find(self, data): curr = self.head while curr and curr.dataval != data: curr = curr.nextval return curr. Code ends.* [Video description ends]

This is one which will search for a particular value among the nodes in the linked list. It is the value to search for which is the data argument of the find function and the value it will return is the node in case the element has been found. It will return None in case of an unsuccessful search. So once again, we make use of the curr pointer and initialize it to the head. And we continue iterating over the linked list until we find a node whose data val field matches the data argument to this function or if the value of the current pointer is null. In the case of a successful search, the curr pointer will point to the node bearing the matching element. And its value will be none in case of an unsuccessful search. In any case once we exit the while loop, we will return the value of our current pointer.

[Video description begins] *In the second code cell, after the line of code: curr = curr.nextval return curr, he adds a code. The code is, code starts: def remove(self, data): curr = self.head prev = None while curr and curr.dataval != data: prev = curr curr = curr.nextval if prev is None: self.head = curr.nextval elif curr: prev.nextval = curr.nextval curr.nextval = None. Code ends.* [Video description ends]

Moving along now to removing elements from the linked list for which we define this remove function, and this is one which will accept a variable called data. And while iterating over the nodes of the linked list, it will see if the value of any of the nodes happens to match data. For the first match which it finds, it will simply remove that node from the linked list and exit from the function. To implement such a removal though, we will be making use of two pointers. There is the curr pointer which would point to the current node we are iterating over. And there is a previous pointer which will point to the node just before it in the linked list. Why exactly do we do this? Well, when we removed one node from the linked list, we will need to make sure that the value of the nextval field of the node previous to the one being removed gets updated to point to the node after the one being removed.

So we initialize the curr and previous pointers. And then following that, we use this while loop to iterate over the list. Once again, we go over the list until we either encounter the end of the list, or we find a matching node. At each iteration of this while loop though, we not only update the value of the current pointer to point to the next node. But we will also make sure that the previous pointer stays one step behind the current pointer by updating its value as well. Now if we find a match in the first step, in which case it is the head node which needs to be removed from the linked list, in that case, the value of the previous pointer will be None. To remove the node at the head of the list, we will need to update the head pointer to point to what was previously the second node in the list.

This of course can be found at curr.nextval. And if it is not the head node which is being removed, we will make use of the previous pointer. And we will update the nextval field of the previous node to point to the node after the one being removed. That value is available at curr.nextval. To ensure that the node being removed is not pointing to any of the existing nodes in the linked list, we update its next val field to none. This concludes the definition of the remove function. And we already have a lot of functions defined within this custom linked list. We're not quite done however, since we will now define a function in order to reverse the ordering of the nodes in our linked list.

[Video description begins] *In the second code cell, after the line of code: curr.nextval = None, he adds a code. The code is, code starts: def reverse(self): curr = self.head prev_node = None next_node = None while curr: nextval = curr.nextval curr.nextval = prev_node prev_node = curr curr = nextval self.head = prev_node. Code ends.* [Video description ends]

So this is an in-place reversal. And in order to implement this, we will need to iterate over each of the nodes in the list, but on this occasion we will need three different pointers. The curr pointer will point to the current node which we are iterating over and this is initialized to the head. And as we saw in the remove function, we will have a pointer to the previous node which is one step behind the current pointer. And we could also make use of a variable which points to the node after the current node. This next node variable though will not be used later on since we will be using another variable to point to the node after the current one.

So we use this while loop in order to iterate over each of the nodes, and we do this until the value of the curr pointer is null. We initialize a variable called nextval and we set this to the node after the current node. Following that, we will update the nextval pointer of the current node to point to the previous node. So what was previously the node before the current node becomes the node after the current node. To continue iterating down the list, we will now update the value of the previous node pointer which now points to the current node. As for the current pointer this would point to what was previously the node after the current node. So at each iteration, we update the nextval field of the current node to point to the previous one. And we make use of both the nextval as well as the previous_node pointers to point to the nodes at either side of the current node.

At the end of the while loop though, we will have reached the last node of the linked list which is now to become the first node. And the value of the previous_node variable will be pointing to that last node. And in order to make that the first node, we will update the head variable of our linked list to point to it. With that done, we have finished implementing the reverse function for our linked list. We're not quite done with reversals though, because we will now implement one more reverse function. And this is to perform the reversal in a recursive manner. We will continue with this implementation in the next video.

Linked Lists: Testing the Functions

[Video description begins] *Topic title: Linked Lists: Testing the Functions. The presenter is Kishan Iyer.* [Video description ends]

It is now time to implement a recursive version of the reverse function which we have just defined. This is called reverse_recursive. Within that, we define a function called recursion, which is the recursive function to perform the reversal.

[Video description begins] *He opens the LinkedList file in the Jupyter Notebook window. He enters a code in the second code cell. The code is, code starts: def recursion(curr, prev): if not curr: return prev nextval = curr.nextval curr.nextval = prev prev = curr curr = nextval return recursion(curr, prev) self.head = recursion(curr=self.head, prev=None). Code ends.* [Video description ends]

The arguments to this include a current node and the previous node. And if the value of the current node is null, it means that we have hit the end of the linked list. In which case, we need to end the recursion, and then you will also return the value of the previous variable. This will point to what was originally the last node in the linked list, and this is soon to be the first node, or the head, of the modified linked list. So what are the steps which occur if you are not at the last node of the linked list? Well, we will once again make use of a nextval variable, which will point to the node after the current node.

And then we will update the value of the nextval field in the current node to the previous node. This is in fact identical to what we had implemented in the non-recursive version of the reverse function. It is not only the updating of the current node, but also updating the value of the pointers which is identical. So the previous pointer moves one step forward by getting updated to curr, and the curr pointer in turn moves one step forward by getting updated to nextval. What we do next is to recursively call this recursion function, where we pass along the value of the curr and previous pointers. So where exactly do we make the first call to our recursion function? This is in fact the last line of code in this function, where we call recursion with the initial values of the current and previous pointers.

The current pointer is initialized to the head of the linked list, whereas the previous pointer is initialized to none. As we saw at the start of the recursion function definition, its return value will be the tail of the original linked list, which needs to become the head of the newly reversed linked list. So we update self.head with this value. So with that done, both of our reverse functions have been implemented. But we are not quite done with the definition of our custom linked list just yet. There is still one more function which we need to include, and this is called count_nodes.

[Video description begins] *In the second code cell, after the line of code: self.head = recursion(curr=self.head, prev=None), he adds a code. The code is, code starts: def count_nodes(self): if (self.head == None): return 0 else: curr = self.head count = 0 while (curr != None): curr = curr.nextval count += 1 return count. Code ends.* [Video description ends]

This, of course, will count the number of nodes in the linked list. We first perform a check to see whether the head variable is none, in which case it means that the linked list is empty, and the count of nodes in this case is 0, which is the value returned. However, in the case of a non-empty linked list, we will first initialize our current pointer to the head and then also initialize a count variable to 0. Following that, we make use of this while loop in order to continue iterating over each of the nodes in the linked list as long as the curr pointer is not none. That is, we have not reached the end of the linked list.

At each iteration, we increment the value of count, and then we also update the current pointer to point to the next node. At the end though, we will have the count of the number of nodes in the count variable, which we will return. Now for this count_nodes operation, we are iterating over the entire linked list because of which the complexity will be Big O (n). An alternative way to implement the count_nodes is to have a variable which is a member of the linked list which will keep count of the number of nodes. Each time there is an insertion of a node that gets implemented by 1.

Each time a node is removed, that gets decremented by 1. And in such a case, if you wish to get the count of the number of nodes, we only need to check the value of the counter, which is a Big O (1) operation. So with that, we have now finally completed the definition of our custom linked list. It is now time to create an instance of this linked list and then invoke each of our functions to ensure that it works exactly as expected.

[Video description begins] *He enters a code in the third code cell. The code is: numbers = LinkedList().* [Video description ends]

This new instance of our linked list is called numbers, and it will be used in order to store string representation of numbers. To see what it's like, we will first initialize it, and then following that, we check the content of numbers.

[Video description begins] *He enters a code in the fourth code cell. The code is: numbers.* [Video description ends]

This will invoke the repr function. And of course, the linked list at this point is empty.

[Video description begins] *He executes the code in the fourth code cell. The output reads: [].* [Video description ends]

Now let us go ahead and test out one of the functions and also add elements to the linked list.

[Video description begins] *He enters a code in the fifth code cell. The code is, code starts: numbers.append("two") numbers.append("three") numbers. Code ends.* [Video description ends]

We will now invoke the append function on two occasions, where we will add the nodes two and three, which are both strings representing numbers. So what are the contents of this numbers object once we invoke the append function? Well, as we expect, it does contain the two strings which we inserted.

[Video description begins] *He executes the code in the fifth code cell. The output reads: [‘two’ -> ‘three’].* [Video description ends]

The first string, which is at the head of the linked list is two and the second one is three. Now it is time to test out the prepend function, which is the one to add an element at the head of the linked list.

[Video description begins] *He enters a code in the sixth code cell. The code is: numbers.prepend(“one”) numbers.* [Video description ends]

So we prepend one to our existing linked list, and then we check out its contents and other values one, two, and three exist in the list in that order.

[Video description begins] *He executes the code in the sixth code cell. The output reads: [‘one’ -> ‘two’ -> ‘three’].* [Video description ends]

We will now invoke the append function on three more occasions, where we will insert the values four, five, and seven.

[Video description begins] *He enters a code in the seventh code cell. The code is, code starts: numbers.append(“four”) numbers.append(“five”) numbers.append(“seven”) numbers. Code ends.* [Video description ends]

And note that the value six is missing, but we will add that in just a little bit. For now though, we expect that four will be added after three, which in turn will be followed by five and seven. And when we view the numbers list that is exactly what we see.

[Video description begins] *He executes the code in the seventh code cell. The output reads: [‘one’ -> ‘two’ -> ‘three’ -> ‘four’ -> ‘five’ -> ‘seven’].* [Video description ends]

All right, it is now time for us to check out the other insert operation which we implemented, and this is the one to insert a node after one containing a specific value.

[Video description begins] *He enters a code in the eighth code cell. The code is, code starts: numbers.add_after(“five”, “six”) numbers . Code ends.* [Video description ends]

Here, we will insert a node containing the value six after the node which contains the value five. So here we invoke the add_after function, and when we view the contents of the list, we can see the strings one, two, three, four, five, six, seven, in that order.

[Video description begins] *He executes the code in the eighth code cell. The output reads: [‘one’ -> ‘two’ -> ‘three’ -> ‘four’ -> ‘five’ -> ‘six’ -> ‘seven’].* [Video description ends]

So the node six was successfully inserted after five and before seven. So all of the functions seem to work well so far. So what about the reverse function, specifically the non-recursive version?

[Video description begins] *He enters a code in the ninth code cell. The code is: numbers.reverse() numbers.* [Video description ends]

So we execute that and check the contents of numbers.

[Video description begins] *He executes the code in the ninth code cell. The output reads: [‘seven’ -> ‘six’ -> ‘five’ -> ‘four’ -> ‘three’ -> ‘two’ -> ‘one’].* [Video description ends]

And the ordering of the nodes has been successfully reversed.

[Video description begins] *He enters a code in the tenth code cell. The code is: numbers.reverse_recursive() numbers.* [Video description ends]

And now, to test out the recursive version of the reverse function, we will invoke reverse_recursive. And when we view numbers, it is back to the original order.

[Video description begins] *He executes the code in the tenth code cell. The output reads: [‘one’ -> ‘two’ -> ‘three’ -> ‘four’ -> ‘five’ -> ‘six’ -> ‘seven’].* [Video description ends]

So both of our reverse functions have been tested and they work correctly. We can continue with the testing of the different functions.

[Video description begins] *He enters a code in the eleventh code cell. The code is: numbers.remove(‘one’) numbers.* [Video description ends]

So this time, we will invoke the remove function, and we will remove the node which contains the value one. This of course is the element at the head of our linked list. So when we run this and view the contents of numbers, we do indeed see that it only contains six nodes and the head of the linked list is now two.

[Video description begins] *He executes the code in the eleventh code cell. The output reads: [‘two’ -> ‘three’ -> ‘four’ -> ‘five’ -> ‘six’ -> ‘seven’].* [Video description ends]

We can go ahead and remove the element six to test out the removal of an element from the middle of the list.

[Video description begins] *He enters a code in the twelfth code cell. The code is: numbers.remove(‘six’) numbers.* [Video description ends]

And the node six has also successfully been removed.

[Video description begins] *He executes the code in the twelfth code cell. The output reads: [‘two’ -> ‘three’ -> ‘four’ -> ‘five’ -> ‘seven’].* [Video description ends]

And the node containing the text five now points to the node containing seven. So what about the count_nodes function? So we invoke that and make sure that it correctly counts the five nodes in our list.

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

And yes, this is exactly what it returns. So we have now successfully implemented a bespoke linked list in Python. And this has included a whole host of functions in order to both work with the linked list and to also perform additional operations such as reversing the ordering of the nodes. With that, we come to the end of this course on implementing data structures in Python.

Exercise: Implementing Data Structures in Python

[Video description begins] *Topic title: Exercise: Implementing Data Structures in Python. The presenter is Kishan Iyer.* [Video description ends]

In this exercise, you will begin by summarizing some of the common operations which can be performed on stacks and also list down their time complexities. During this course, we not only implemented the various operations for stacks, we also saw what the complexity of those operations will be. And you will need to recollect exactly what those are. Next, you will list the situations when adding or removing elements from a stack or a queue and throw an exception.

During this course, we covered the case when such exceptions can be thrown, and we also briefly discussed how we can avoid such exceptions. So you will need to identify exactly when these errors are thrown and also how we can prevent them. Each of these topics were covered during this course. So please pause this video and then spend some time to do this exercise on your own.

[Video description begins] *Solution. Common Stack Operations.* [Video description ends]

We begin with the first task in the exercise, where you needed to identify some of the common stack operations and also note down the complexity of those operations. Once you start off with an empty stack, the most likely first course of action for you is to PUSH an element onto it. Specifically, you will be pushing an element to the top of the stack. And because you're always referencing the top, you don't need to iterate over the contents of the stack, which is why this operation will run in constant time and have the complexity of Big O(1). The converse operation to a PUSH is the POP, and this is where you remove an element, once again, from the top of the stack. When popping an element from the stack, you always remove it from the top.

And no matter what the size of the stack, the steps that you need to perform remain exactly the same, which is why the POP operation also runs in constant time. Another thing you're likely to do very often when working with a stack data structure is to take a PEEK at the element which is on top of the stack. Once again, you're referencing the top of the stack, which is why no matter what the size, this operation will run in constant time. Now, many different stack implementations will have a maximum capacity for the stack. And no matter what the size of the stack, you may have a function in order to check whether the stack is currently empty.

That is, it does not contain any elements. And in the case the stack does have a maximum capacity, you may also implement a function to check whether the stack is full. Both of these operations can be defined in such a way that they run in constant time. This will involve the use of variables, which keep a track of not only the maximum capacity of the stack, but also the number of elements which have been loaded into it.

[Video description begins] *Add and Remove Exceptions.* [Video description ends]

For the next task in the exercise, you needed to identify specific situations where adding elements to, or removing elements from, stacks and queues can generate exceptions. Now, an exception can be thrown when adding elements when the stack or the queue is full. This of course applies when each of these data structures has been set a maximum capacity. Conversely, an exception is also thrown when we try to remove elements from a stack or a queue when they happen to be empty. And we did see examples of this during the course.

Now how exactly can we avoid such exceptions? Well, we can always perform a check before we either add elements or remove them from the stack or a queue. So if we implement an ISFULL function, we can run this function to check for a full queue before we add any elements to it. And we can also make use of an ISEMPTY function in order to check for an empty queue before we trigger a removal of any elements.