Conditional Statements & Loops: Advanced Operations Using for Loops in Python

Explore how iterating over elements using for loops can be controlled using the break and continue statements in Python. Creating sequences from other sequences using comprehensions is also covered in this 9-video course. Key concepts covered here include how to terminate a for loop when a specific condition is met using the break statement; learning how the break statement affects the code in the else block of a for loop; and observing how to skip an iteration of a for loop when a specific condition is met using the continue statement. Next, learn how to use the continue statement along with the break statement within the same for loop; learn the fact that no action is performed under specific conditions by using the pass statement; and create a list out of the contents of another list using a comprehension. Finally, you will learn about conditions in list comprehensions in order to filter elements used in the source list and to define values in the newly created list.

Table of Contents

Course Overview

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

Hi and welcome to this course, Advanced Operations with For Loops 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. For all its popularity, Python is at its heart a programming language just like any other. A Python program, just like a program in any other language, can be expressed as a combination of different control structures of the following three types.

The first of which is a default sequential control structure in which statements execute one after another. A second control structure is one in which conditions are evaluated before deciding what the program does next. And finally, there are iterative or repetitive control structures in which statements are repeatedly executed until a defined terminating condition is met. This course covers the advanced operations which can be performed with for loops. It is purely hands-on and includes several examples for you to gain a firm understanding of these control structures in Python. We begin by exploring the break statement, which is used to terminate a for loop once a specific condition is met.

We then take a look at how you can skip actions in specific iterations of the for loop by using the continue statement and how you can include placeholders in your code using the pass keyword. Finally, we cover how you can create lists from the contents of other lists in a simple and elegant manner using comprehensions. By the end of this course, you will be quite the expert on Python for loops. You will know exactly how to control their execution using the break and continue statements and how to quickly create list, tuples, and other sequences using comprehensions. You can then move on to exploring other topics in Python programming, such as functions and classes.

Introducing the break Statement

[Video description begins] *Topic title: Introducing the break Statement. The presenter is Kishan Iyer.* [Video description ends]

All the for loops which we have implemented so far in this course, have one thing in common. We have iterated over each and every element within the range specified, whether it is a range defined using a range function, or if it is a list or a tuple of values. But what if you want to terminate the loop somewhere in the middle based on a condition? Let's just say you are looping through a list of elements, and you're searching for one which contains a specific value.

[Video description begins] *The BreakStatementInForLoop page is open in the jupyter notebook. It is divided into three parts. The first part is a menu bar. It includes File, Edit, Insert, and View options. The second part is a toolbar. The third part is a content pane.* [Video description ends]

Once you find that value, you don't need to continue your search with the other elements and you can simply terminate the search and break out of the for loop. And that is exactly what the break statement will help you accomplish. In this demo, we will see exactly how that works. And for that I have created a new Jupyter notebook which I have named BreakStatementInForLoop. So let us begin coding for this demo. So first, we will implement a for loop which iterates over each of the characters in the word python.

[Video description begins] *He enters the lines of code, code starts: for letter in "python": if letter == "o": break print(letter). Code ends.* [Video description ends]

And then, we will specify an if condition within it. So if we find a character which matches the letter o, then we will break out of the loop. For that we make use of the break keyword.

[Video description begins] *The output reads: p y t h.* [Video description ends]

And what this means is that the for loop will just terminate at this point and no more iterations will be performed. In order to visualize the letters which have been processed in each iteration, we include this print statement at the end, which is outside the scope of the if condition, but within the for loop. So we run the cell, and what we see from the output was that the print statement was executed for each of the letters p, y, t, and h.

[Video description begins] *He enters the lines of code, code starts: for letter in "python": print(letter) if letter == "o": break. Code ends.* [Video description ends]

But we did not get to the print statement for either the letter o or for n. What is clear here, is that all of the iterations of the for loop were not completed. But what is not entirely clear here is, at what point the loop dominated? Did the if statement ever evaluate the letter o, or did the loop somehow terminate after printing out the letter h?

Well, to clarify, we will slightly modify this for loop, which we implemented, except this time, we will print out the letter before evaluating letter using our if statement. So if the letter o gets printed out in this case, but the letter n does not, then we can be confident that it was the break statement which caused the loop to terminate. And we run this, and that is precisely what has occurred here. So this for loop is effectively a search for the letter o in the string python.

[Video description begins] *The output reads: p y t h o.* [Video description ends]

Once we found the letter, we don't really need to continue with the search, which means that we can terminate the for loop. The way to do that in Python is to make use of the break statement, which will effectively break out of the for loop and make sure that none of the other elements will be processed. Now, how exactly does this affect some of the other things we have covered previously in this course? You will recall that we covered the use of an else statement along with a for loop, just as a reminder.

[Video description begins] *He enters the lines of code, code starts: for char in 'icecream': if char == 'c': print ('The letter %s was found in the string' % char) break else: print('The letter %s does not exist in the string' % char). Code ends.* [Video description ends]

When you include an else block with a for loop, it means that once all of the iterations of the for loop have completed, then all of the code defined within the else block will be executed. However, if the for loop were to terminate prematurely for some reason, whether due to an error or whether due to the invocation of a break statement, then the code within the else block will not be run. So in this particular example, we iterate through each of the characters in the word icecream, and if we find the character c, then we will print out that the letter c was found, and then we will break out of the loop.

Now, if we were to iterate over each and every element within the for loop, this means that the letter c has not been found. And in this case, we want the print statement defined within the else block to be executed, that the letter does not exist within the string. So we should expect here, that running this will mean that the code within the else block will not be executed. And that is indeed the case, the letter c was found in the string, which means that the for loop terminated before all of the iterations were complete.

[Video description begins] *He runs the code and the output reads: The letter c was found in the string.* [Video description ends]

And because of that, the else block associated with a for has not been executed. Now you can replace the word icecream with another word which does not contain the character c, in which case you will get the message that the letter does not exist within the string. But for now, we will move on to one more example, where we have a list of five different integers.

[Video description begins] *He enters the lines of code, code starts: for num in [3, 11, 22, 35, 90]: print (num) if (num == 22): print ("The number %d has been found" % num) print ("Terminating the loop...") break. Code ends.* [Video description ends]

We will iterate over each of these integers and then print out the value, and then we will perform a check to see whether that integer equals 22. If it is, we will issue these two print statements to convey that the number has been found and that we are terminating the loop. And the actual termination of the loop will take place due to the break statement.

[Video description begins] *The output reads: 3 11 22 The number 22 has been found Terminating the loop....* [Video description ends]

And when we run this, we can see that in the first iteration, it was a number 3 which was evaluated. Then it was 11, and then the number 22 was found. At this point the two print statements were executed, and then nothing happened beyond that. The numbers 35 and 90 were never really iterated over, because the for loop had terminated once the number 22 had been found, and the break statement was executed. We will now create one more for loop, this time by making use of the range function.

[Video description begins] *He enters the lines of code, code starts: for i in range (5): if i == 4: break print(i). Code ends.* [Video description ends]

Here, we check whether the value in our iteration happens to be 4. And if it is, we will break out of the loop and otherwise we will print out the value of i in the iteration. And when we run this, we get the values 0, 1, 2, and 3.

[Video description begins] *The output reads: 0 1 2 3.* [Video description ends]

The for loop did go into the next iteration where i was equal to 4, but then the condition defined within our if statement evaluated to true, in which case the break statement was executed. At that point, we broke out of the loop and all of the code following the break within the loop itself was not executed. So when the break statement has encountered, it does not mean that the current iteration will be seen through to completion, but that the loop will terminate at exactly that point. Moving along to another for loop, where we go over the range of values between 1 and 9, unless that is we encounter a value which is greater than or equal to 5.

[Video description begins] *He enters the lines of code, code starts: for i in range (1, 10): if i >=5: break print(i, "is less than 5"). Code ends.* [Video description ends]

Once such a value if encountered, we break out of the loop. However, if the value does not satisfy this if condition, then we will print out the value of i and say that it is less than 5. And when we run this, sure enough, the print statement does execute for the values 1, 2, 3, and 4.

[Video description begins] *The output reads: 1 is less than 5 2 is less than 5 3 is less than 5 4 is less than 5. Code ends. He highlights the code line: for i in range(1, 10):.* [Video description ends]

But even though there are values greater than 5 in our range, we just break out of the for loop when the value of i equals 5, and then i never gets to take on the values of 6, 7, 8, and 9. We now write one more for loop, where we iterate over the range of integers from 0 through 9.

[Video description begins] *He enters the lines of code, code starts: my_list = range(10) for num in my_list: if num ** 2 > 20: break print(num, "square", num ** 2). Code ends.* [Video description ends]

And then we will evaluate whether the square of that integer happens to be greater than 20. If so, we will break out of the for loop, and if not we will print out the value of both the number as well as its square. And sure enough, this loop breaks once it encounters the value 5, whose square is greater than 20.

[Video description begins] *The output reads: 0 square 0 1 square 1 2 square 4 3 square 9 4 square 16.* [Video description ends]

The break Statement and the else block

[Video description begins] *Topic title: The break Statement and the else block. The presenter is Kishan Iyer.* [Video description ends]

So we have now seen that the break statement is very useful when we're effectively performing a search within a sequence of elements over which we are performing iterations.

[Video description begins] *The BreakStatementInForLoop page is open in the jupyter notebook.* [Video description ends]

We can try to visualize that by playing a little game.

[Video description begins] *He enters the lines of code, code starts: places = ["New Zealand", "Norway", "Botswana", "Zimbabwe", "Uzbekistan", "Paraguay"] villain_at = "Mali" for place in places: if place == villain_at: print("The Villain has been captured!") break else: print("The villain got away :("). Code ends.* [Video description ends]

Let's just say we have a list of places and these happen to represent countries from all of the world and then we are in search of a villain. Our task is to visit each of the countries in our list of places and then see if that is where the villain is. If you find the villain, we can then stop our search and then say that the villain has been found. But if you have visited each and every place and then not found the villain yet, it means that the villain has gotten away. So this has some similarity to the game where in the World Is Carmen Sandiego from the 1980s and 90s.

So you can say here that in our list of places, the country Mali is not present, but that is exactly where the villain is. So we will still carry out the search, and for that we implement a for loop, in which we will iterate over each individual place in our places list. If the place we are in happens to be exactly where the villain is at, then we will print out that the villain has been captured. And then end our search by issuing the break statement and breaking out of the loop.

However, if you have gone through every iteration of the for loop, it means that the villain has not been found. In which case, we can make use of the else block for the for loop, and we will print out that the villain has gotten away. So when we run this, as expected, we get the message that the villain has escaped us.

[Video description begins] *The output reads: The villain got away :(*. [Video description ends]

However, if we rebuild this for loop with just 1 minor modification where we adjust the location where the villain is from Mali to Norway. We can rerun this for loop and on this occasion, we get the message that The villain has been captured.

[Video description begins] *The output reads: The Villain has been captured!*. [Video description ends]

You will now know that we only ever got to the second iteration of the for loop since that is the location where the villain was found. Thanks to our use of the break statement, we would stop our search once a specific condition was met. While you may find this to be a somewhat contrived example, we will now take a look at another instance where this early termination of a loop can be very useful.

[Video description begins] *He enters the lines of code, code starts: num = int(input("Enter a number: ")) for i in range(2, num//2): if num % i == 0: print("It is not a prime number") break else: print(num,"is a prime number"). Code ends*. [Video description ends]

So consider that we wish to evaluate whether a given number happens to be a prime. The way to do this is to divide the number using every integer from 2 until half the value of that number. And if our input number happens to be divisible by any of those integers within that range, then it is not a prime number. On the other hand, if the input number is not divisible by any of those numbers, then it is a prime. So this logic fits perfectly with what we have seen so far with the use of a for loop along with a range function, and also the use of the else block to go with the for loop.

We start off by asking the user for an input and this will be the number which you will check for whether it is a prime or not. So this is assigned to the variable num and then we define our for loop. Here, we need to iterate over all of the integers between 2 and half the value of num, and check whether any of those integers happened to be a factor of num. For this we make use of the range function, whose lower bound is 2 and the upper bound is num divided by 2. And note here we're making use of the double slash so that the quotient is an integer.

So for each of the values of i in this range of integers, we check whether num modulus i is equal to 0. If it is, it means that i is a factor of num, in which case num is not a prime number. So the first time we find such a value of i, we can print out this message and simply break out of the loop. We don't care if there are any other values of i which are a factor of num as long as there is 1 in this range, it means that the number is not a prime. On the other hand, if you go through the entire range of integers and don't find a single number which happens to be a factor of num, it means that the number is a prime number.

And it will also mean that we have performed every single iteration within our range without encountering a break statement. In this case, our code will end up going into the else block associated with a for loop. And the print statement within that that the number is a prime number will be executed. So let us go ahead and run the cell. When prompted for an input, I will first enter the value 6 and sure enough, it generates the output that this is not a prime number.

[Video description begins] *The output reads: Enter a number:*. [Video description ends]

In fact, at the very first iteration when it will divide the number 6 by 2, our if condition will be satisfied, and we will break out of the for loop. So let us go ahead and run the cell once more. But on this occasion, I'm going to specify an input of 7. And sure enough, this returns that this is in fact a prime number. So this is a good illustration of how we can make use of the range function, the break statement, and also the else block along with a for loop.

And we have also observed that the use of the break statement here makes the code execution very efficient. Since we don't need to go over each and every iteration, we can now take a look at 1 more example where the combination of the range break and else blocks are all used together.

[Video description begins] *He enters the lines of code, code starts: for i in range(3): password = str(input("Enter a password:)) if password == "secret": print ("you guessed the password:)") break else: print ("3 incorrect password attempts"). Code ends.* [Video description ends]

So in this case, we want to prompt the user to enter a password a maximum of 3 times. If anyone of those attempts results in the correct password being typed, then the user will no longer be prompted for an input. However, if the user does type the password wrong three times, then we will print out the fact that there were 3 incorrect password attempts. So in order to issue a prompt a maximum of 3 times, we make use of a for loop, which uses the range function. And the range of course is 3. So here, we will prompt the user for an input, and this is where the user enters the password.

And this value entered is assigned to the variable password. We then check whether the user has entered the correct password. The password in our case is secret. So if the variable password matches secret, then we print out that the user has guessed the password correctly. And in this case, we no longer need to prompt the user for the password again. So we just issued the break statement and break out of the for loop. On the other hand, if there have been 3 unsuccessful attempt to the password, then the break statement will never be executed. In which case, the code execution will end up at the else block.

This is where we will print out the fact that there have been three incorrect password attempts. So let us go ahead and execute this for loop. So we are prompted for a password. I type in the string secret. And sure enough, within the first iteration of the for loop, we encounter the break statement and we are no longer prompted for the password.

[Video description begins] *The output reads: Enter a password: secret you guessed the password :).* [Video description ends]

You can use the same for loop and then enter the wrong password. And then ensure that you do get to the else block, if you enter the wrong password on 3 occasions.

The continue Statement: Part 1

[Video description begins] *Topic title: The continue Statement: Part1. The presenter is Kishan Iyer.* [Video description ends]

When we started off this course on for loops, we examined how we can iterate over an entire collection and then perform a certain set of operation on each of the elements in that collection.

[Video description begins] *The ContinueStatementInForLoop page is open in the jupyter notebook.* [Video description ends]

We then explored the break statement, where we can check for a specific condition during an iteration. And if the condition is met, it means that our work is complete and we can break out of the for loop. Until that condition is met though, we will continue performing certain steps within the body of the for loops. The break statement though, represents a somewhat extreme course of action. Consider the case where we are iterating over a sequence.

And if we happen to find a value, we don't want to break out of the for loop, but we simply wish to skip that particular iteration and then move on to the next one without executing any of the actions defined within the for loop body. This is precisely where the continue statement comes into the picture and that is what we will be exploring in this demo. Once again, I have created a new Jupyter notebook for this which is called ContinueStatementInForLoop. All right, let us now go ahead and then demonstrate exactly how a continue statement will work.

[Video description begins] *He enters the lines of code, code starts: for letter in "string": if letter == "i": continue print(letter) print ("The end"). Code ends* [Video description ends]

So we create a for loop once again, and this time we will be iterating over the character which make up the word string. There is a default course of action which we need to perform for each of the characters, and that is to print out the character itself. So we include a print statement within a for loop body. But before that, we define a condition where we will not take the default course of action, but will instead skip that iteration and then move ahead to the next one in the for loop. So within the if statement we check whether the character is equal to i, and if it is, we invoke the continue statement.

What does this do? It ensures that no more code is executed in the current iteration of the for loop. And then you move on to the beginning of the next iteration. So any code within the for loop body which occurs after the continue statement will not be executed once the continue statement is invoked. In our case, if the letter is equal to i we will not print the character. And at the end of all the for loop iterations, we will print out the message, The end. So let us see what happens when we run the cell.

[Video description begins] *The output reads: s t r n g The end.* [Video description ends]

So you can see that each of the character which make up the word string are printed out, except for the letter i. So consider the case of the letter s in our first iteration, where we enter the for loop, and then we perform the condition check in the if statement. Since the value of letter in that case was not equal to i, we did not invoke the continue statement, and instead the code went ahead and then printed out the letter. The same thing occurred in the case of the characters t, r, n, and g as well. But in the case of i, the condition in the if statement evaluated to true, and then the continue statement was invoked.

For that iteration, the code execution never got to the print command and instead went over to the beginning of the next iteration to evaluate the letter i. So you can say that the continue statement is somewhat similar to the break statement in that it introduces some disruption in the flow of the loop. While the break statement terminates the loop altogether, the continue statement only terminates the current iteration. So this is an important point of distinction. Let us go ahead now and then examine one more use of the continue statement. Here within the body of our for loop we include an if else block.

[Video description begins] *He enters the lines of code, code starts: my_string = "Students" for letter in my_string: if (letter == "e" or letter == "s"): continue else: print(letter). Code ends.* [Video description ends]

So here, there is a condition in our if statement, where we check for the letter e or the letter s. So note here that this is a lowercase s. If this condition is satisfied, then we continue to the next iteration of the for loop and otherwise we print the value of letter for that iteration. So note here, that unlike in the previous for loop, the printing of the value of letter is not considered the default action because it is now part of the else block.

However, the inclusion of the continue statement within the if block is a representation that we don't need to take any action for this iteration. So let us go ahead and run this, where we will iterate over the character in the string Students, which begins with an uppercase S. And you can see that all of the characters are printed out, except for the lowercase s at the end and the lowercase e, a little earlier within the string.

[Video description begins] *The output reads: S t u d n t*. [Video description ends]

Now, let us make use of another if else statement within the for loop, but this time, we have actions within the if block. We include a continue statement within the else block.

[Video description begins] *He enters the lines of code, code starts: for i in range (1, 10): if i % 2 == 0: print ("Is %s is an even number? " % i) else: continue print("Yes"). Code ends*. [Video description ends]

And then we have a default action for the for loop, where we print out the message, Yes. So here we are iterating over all of the numbers between 1 and 9. And in the if statement, we check whether that number is divisible by 2. If it is, we ask the question whether it is an even number, but then don't answer the question right away. Within the else block, we simply invoke the continue statement, and the default course of action in our for loop is to answer the question whether the number is even with a Yes. So let us go ahead and run this.

[Video description begins] *The output reads: Is 2 is an even number? Yes Is 4 is an even number? Yes Is 6 is an even number? Yes Is 8 is an even number? Yes*. [Video description ends]

So let us examine the output now. For the first iteration, the value of i will be 1 and then the condition in the if statement will evaluate to false. So we don't invoke the print statement within that. Since the if condition evaluated to false, we will move ahead to the else block and then the continue statement will be invoked. That particular iteration will then cease and then we move ahead to the next iteration where the value of i will be 2. So the continued statement will be invoked for every odd number in our range.

However, in the case of even numbers, for example 2, the if condition will evaluate to true and print statement within that will be executed. And the iteration will not just end there, but will instead move to the print statement at the very end and print out Yes. And this applies to every even number as well. We can now perform a slight modification on this for loop, where we once again iterate over the range of numbers from 1 through 9.

[Video description begins] *He enters the lines of code, code starts: for num in range(1, 10): if num % 2 == 0: print(" Found an even number:" , num) continue print(num)*. [Video description ends]

And once again you perform a check to see whether the number at any iteration is divisible by 2. In this case however, when the condition is satisfied, we print out the fact that it is an even number and then we invoke the continue statement. So this is what happens when the number is even, but if the number is odd, then the if condition will evaluate to false and we will just print out the number directly. So we can go ahead and run this for loop.

[Video description begins] *The output reads: 1 Found an even number: 2 3 Found an even number: 4 5 Found an even number: 6 7 Found an even number: 8 9*. [Video description ends]

And you can see here that thanks to the continue statement, the last print statement within our for loop body is never invoked for even numbers. So this is a little different from our previous invocations of the continue statement. On those occasions, once a condition was satisfied, we did not invoke any actions and we simply called the continue statement directly. In this case though, we have one course of action for even numbers, and then we trigger the continue statement, and then a default course of action for all other numbers.

The continue Statement: Part 2

[Video description begins] *Topic title: The continue Statement: Part2. The presenter is Kishan Iyer.* [Video description ends]

We now introduce one more variation in our for loop. On this occasion, we iterate over the range of value from 1 through 11.

[Video description begins] *The ContinueStatementInForLoop page opens in the jupyter notebook. He enters the lines of code, code starts: for i in range (1, 12): print() print ("Before the continue statement", i) if i % 3 == 0: continue print("After the continue statement", i). Code ends.* [Video description ends]

And then we include a set of actions within the for loop body, which occurs before the evaluation of a condition. So here, we print a blank line. And then follow that up by printing out the message that we are now Before the continue statement. We then check whether the value of i in this iteration is divisible by 3. And if it is, we invoke the continue statement and then move ahead to the next iteration. On the other hand, for numbers not divisible by 3, we will also print out the message After the continue statement. So in this case, the first two print statements will be executed for every single iteration. Whereas the last print statement will only be invoked for those numbers which are not divisible by 3. We can go ahead and execute this and see exactly what the output is like.

[Video description begins] *The output displays the Before the continue statement and After the continue statement for the numbers which are not divisible by 3 and only the Before the continue statement for the numbers which are divisible by 3.* [Video description ends]

So you can see here that both the before and after print statements have been executed when i = 1 or 2, but only the first of those print statements has been invoked when i = 3. So in this for loop, we have a default set of actions for every iteration. And then we have one course of actions if the number is divisible by 3. And another course of action when the number is not. We can take a look at some more examples in order to get familiar with the use of the continue statement.

[Video description begins] *He enters the lines of code, code starts: num_days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31] months= ["Jan", "Feb", "Mar", "Apr, "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"] new_dict = dict() for i, j in zip(months, num_days): new_dict.update({i:j}) if new_dict[i] < 29: continue else: print(i, "month has", new_dict[i], "days"). Code ends.* [Video description ends]

So we start off here with two different lists. The first one contains a set of integers, which represent the days of the month for each of the month from January to December. And another list called months which contains the string representation for each of the months. We then initialize a dictionary called new_dict. And into that dictionary we will be loading a set of key and value pairs. Where the key will be the string representation of each month, and the corresponding value will be the number of days in that month. Now, how do we load values into this dictionary?

Well for that, we make use of the zip function in Python, which will combine two lists in order to form a list of tuples. By zipping the lists, months, and num_days, both of which have the exact same length. We will be creating a list of tuples where within each tuple, we will have the name of the month and the number of days in that month. We set up a for loop in order to iterate over each of the tuples in that list of tuples. And then the individual elements within the tuple will be assigned to variables i and j for each iteration. Now, the first thing we do within each iteration of the for loop is to add the values i and j into a dictionary, where i becomes the key and j is corresponding value. So this is how we load our dictionary with the name of the month along with the number of days in the month as its value.

Make note of the use of the update function for a dictionary here. Where if the key value pair happens to be new, then it will be added to the dictionary. And if the key already exists, then its corresponding value will be updated. So once this key value pair has been added to the dictionary, we check if the number of days in that corresponding month happens to be less than 29. That is, if it is February, then we will invoke the continue statement and we'll skip over to the next iteration in the for loop. However, if the month is not February and the number of days is greater than or equal to 29, in that case we will invoke this print statement. Where we print out the name of the month and its corresponding number of days. So let us go ahead and run this. And as you would expect, we get a print statement invocation for every one of the months except for February.

[Video description begins] *The output displays the number of days in each month except February.* [Video description ends]

To round off this demo on the use of continue statement, we will see how we can make use of both the continue as well as break statements within the same for loop.

[Video description begins] *He enters the lines of code, code starts: animals = ["Deer", "Rabbit", "Lion", "Dog", "Cat", "Elephant", "Giraffe"] for pet in animals: if pet == "Lion": continue if pet == "Elephant": break print (pet). Code ends.* [Video description ends]

Here we will be iterating over a list of animals and we will be considering each of these animals as exotic pets. So if in our iteration we come across a pet which happens to be a Lion, then we will skip over to the next iteration without invoking the print statement at the bottom. On the other hand, if we encounter the pet, Elephant, then we will break out of the loop entirely. And not only will the print statement at the bottom not be executed, but none of the further iterations in the for loop will take place either. So we can execute this in order to illustrate the differences between continue and break. And this is the output.

[Video description begins] *The output reads: Deer Rabbit Dog Cat.* [Video description ends]

You can see that for the first two iterations, Deer and Rabbit, the print statement at the bottom has been invoked. In the case of Lion the continued statement was invoked, but this did not break us out of the loop. It was only that particular iteration which terminated and the loop itself did continue execution. And this is why the animals Dog and Cat were printed out. However, when the animal Elephant was encountered in this for loop, we invoke the break statement. And not only did that terminate the current iteration, but it also entirely terminated the for loop. So with that, you should have a fairly good understanding of not only for loops by now, but also how you can control the flow of the for loops by making use of the break as well as continue statements.

The pass Statement

[Video description begins] *Topic title: The pass Statement. The presenter is Kishan Iyer.* [Video description ends]

Previously in this learning path, we have covered different conditions which can be evaluated at each iteration of the for loop.

[Video description begins] *The PassStatementInForLoop page is open in the jupyter notebook.* [Video description ends]

And actions can be taken according to the outcome of that evaluation. For example we have seen that we can either invoke a break or a continue statement or we can define our own set of actions with code. However, there will be some situations where you have a default set of actions, but if a specific condition is met, you want to do nothing. So this is where you can make use of the pass statement in Python. Let us see exactly how this will work. So I have created a new Jupyter notebook for this demo, which I have called PassStatementInForLoop. And we'll go ahead and paste a new for loop within the first cell which includes this pass statement.

[Video description begins] *He enters the lines of code, code starts: for letter in "string": if letter == "r": pass print (letter). Code ends.* [Video description ends]

So once again we will iterate over each of the characters in the word string. And then we perform an evaluation using an if statement where we check whether the letter in that iteration is equal to r. If it is, we invoke the pass statement. And the default course of action in every single iteration of the for loop is to print the value of letter. So what happens when we execute this? So each of the letters in the word string have been printed out.

[Video description begins] *The output reads: s t r i n g.* [Video description ends]

So it looks like nothing has really taken place by the introduction of the pass statement. And in fact, this is precisely what the pass statement is used for. Where it is used to denote the fact that nothing needs to be done. However it makes sense to include the pass statement from a syntax point of view to explicitly state that there is no action required for that condition. Now why exactly might this be useful? So to consider that, we will build upon our example.

[Video description begins] *He enters the lines of code, code starts: for letter in "string": if letter == "r": pass print("This is the pass block") else: print("Current Letter :", letter) print ("out of the loop"). Code ends.* [Video description ends]

So we once again introduce a for loop to iterate over the letters in the word string. And again we have an if condition where we check for the letter r, and we include the pass statement within this block. And we also include a print statement where we say that we are now in the pass block. We include an else condition within our for loop where we will print out the value of the Current Letter. So in this case when we iterate over the letters in the word string, all of the letters other than r will be printed out. However when the letter r is encountered, we should execute the print statement that we are within the pass block. So we execute this and we can see that that is exactly what occurs.

[Video description begins] *The output reads: Current Letter : s Current Letter : t This is the pass block Current Letter : i Current Letter : n Current Letter : g out of the loop. Code ends.* [Video description ends]

The inclusion of the pass statement does not prevent the code execution moving to the print statement defined immediately after it within the pass block. And this example also reinforces the notion that the pass block does exactly nothing. But where exactly can this be useful? So in this example, we will see how the pass statement helps from a syntax standpoint.

[Video description begins] *He enters the lines of code, code starts: for i in range(10): if (i % 2 == 0): pass else: print(i). Code ends.* [Video description ends]

Here, we iterate over the integers from zero through nine. And if the number happens to be divisible by two, that is if it is an even number, we do nothing. And if the number is not even, then we print out its value. So in this case, by including the pass statement, we explicitly state that nothing needs to be done if the number is even. Though this doesn't really affect execution, for anyone reading the code, it becomes clear that when this condition is met, nothing needs to be done. And it is also a lot easier to read than if we simply left the if block blank. So we will execute this just to test out this for loop, and it behaves exactly as we expect it to.

[Video description begins] *The output reads: 1 3 5 7 9.* [Video description ends]

Now to illustrate another case where this might be a little useful, we will just build upon this particular for loop example.

[Video description begins] *He enters the lines of code, code starts: odd_count = 0 total_count = 0 for i in range(10): if(i % 2 == 0): pass else: print(i) odd_count += 1 total_count += 1 print("Count of odd numbers: ", odd_count) print("Total count of numbers: ", total_count, "\n"). Code ends.* [Video description ends]

So now let us say we wish to implement this for loop in order to count not just the total number of integers in a range. But also the total number of odd numbers within that range. To keep tracks of these counts, we create two variables called odd_count and total_count which are initialized to 0. We use the for loop in order to iterate over the integers from zero through nine. Once again, we use an if condition in order to check for an even number. And if it is even, we simply say that nothing needs to be done by invoking the pass command.

On the other hand, if the number is not even, we invoke a print statement to print out the number itself and also increment the value of the odd_count. Now whether or not the number is even, we will increment the value of total_count by one. And at each iteration of a for loop, we will maintain a running counter of the odd_count and total_count and then print them out for the users to view. So let us go ahead and execute this for loop.

[Video description begins] *The output displays the count of odd numbers and the total count of numbers. When the total count of numbers is odd, then the incremented value is printed for the count of odd numbers and when* *the total count of numbers is even then the same value is printed for the count of odd numbers*. [Video description ends]

We can see here that in the case of the even numbers, only the final two print statements are executed. In the case of odd numbers however, there is an additional print statement and the value of odd_count is also incremented. So once again, anyone looking at the code will be able to see that nothing needs to be done if the number evaluates to an even number. Now, in the case of a simple if and else block, the inclusion of the pass statement may not seem particularly useful. Especially since we can make use of a single if statement to check for an odd number and then take action if that is the case. However, the use of pass becomes more apparent in this specific example.

[Video description begins] *He enters the lines of code, code starts: my_str = input ("Input a string: ") digits = 0 letters = 0 for i in my_str: if i.isdigit(): digits = digits + 1 elif i.isalpha(): letters = letters + 1 else: pass print("Digits = ", digits) print("Letters = ", letters). Code ends*. [Video description ends]

Here we accept an input string from the user and store it into a variable my_str. And then we will count the number of alphabets and digits in that input string. So for that, we maintain these counters of digits and letters which are initialized to zero. And then use a for loop in order to iterate over each of the characters in my_str. First, we check whether the character is a digit by invoking the isdigit function. And if it is, we increment the value of digits. We then check whether the character is an alphabet for which we make use of an elif block. And then we increment the value of letters if it is an alphabet.

Now what if this character is neither a digit nor an alphabet? Well then our code execution will flow to the else block, and we say that in this occasion, we will do nothing. And this course of no action is denoted by the invocation of the pass statement. At the end of the for loop, we will print out the number of digits and the number of letters in the input string. So let us go ahead and run this. I will now enter the string python#$3, which includes two characters which are neither alphabets nor digits. These are the hash or pound and dollar symbols. So let us see what happens when this for loop executes for the string. So the number of digits and the number of letters have been calculated correctly.

[Video description begins] *The output reads: Input a string: python#$3 Digits = 1 Letters = 6*. [Video description ends]

Within the for loop when the value of i was equal to the pound symbol or the dollar sign, no action was taken. And this fact becomes clear to anyone reading the code when they see the inclusion of the pass statement. One more use for the pass statement is that it can serve as a placeholder. So if there is some code which you would like to implement later on but you would like to place some kind of marker for it for the moment. Then you can use a pass statement.

Introducing Comprehensions

[Video description begins] *Topic title: Introducing Comprehensions. The presenter is Kishan Iyer.* [Video description ends]

We have now explored different uses of for loops in Python.

[Video description begins] *The Comprehensions page is open in the jupyter notebook*. [Video description ends]

But there is another very useful and rather cool feature, which allows us to create lists out of the contents of another list, by making use of a for loop. While intuitively, you will know that it should be possible to do that with for loops. The cool thing about Python with respect to other programming languages is how these lists can be created from other lists using a very brief and elegant syntax. This feature is known as list comprehension. So I have now created a new Jupyter notebook which I have named Comprehensions. To see exactly how this works, we will go ahead and create a sequence of numbers.

[Video description begins] *He enters the line of code: num = range(10).* [Video description ends]

Specifically, we will create a sequence of the numbers from zero through nine using the range function. And this range is available in this variable num. So what if we wanted to create another list using the contents of num? Let's just say for each element which is present in num, we want to create another list which contains the square of those elements. So you can easily make use of a for loop which iterates over each element in num. And then append the square of each of those elements into a new list. However, this can be done using a much shorter and more easy to follow syntax by using list comprehensions. So here we initialize a list by specifying the square brackets. And within that we define exactly what this list should contain.

[Video description begins] *He enters the line of code: [x * x for x in num].* [Video description ends]

So we say that this list should have x square for every x in num. And note the syntax here. We initialize what is effectively a for loop by saying for x in num. The operations which need to be performed for each x in num is defined just before the appearance of the for keyword. In our case, our operation is to perform a square of x. So this syntax will ensure that we create a new list which contains the same number of elements as the sequence num. However, for every element x present in num, this list will contain x square. So we run the cell and a list of squares have been produced.

[Video description begins] *The output reads: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81].* [Video description ends]

And for this, we only needed one line of code. So while we have worked with integers in this example, let us take a look of working with strings.

[Video description begins] *He enters the lines of code: letters = [letter for letter in "anxiety"] print (letters).* [Video description ends]

Here we create a new list containing the characters in the word anxiety by iterating over each of the letters in that word. So we set up a list by specifying the square brackets. And then within that, we set up iterations over the letters which make up the word anxiety. And for each letter, the contents of this list we are about to create will contain the letter itself. In this case, we are not transforming each of the elements we are iterating over in any case, we're just using them as is.

[Video description begins] *The output reads: ['a', 'n', 'x', 'i', 'e', 't', 'y'].* [Video description ends]

So when we run this, we can see that all the letters of the word anxiety have been loaded into a list. We can do one more list comprehension now.

[Video description begins] *He enters the lines of code, code starts: list_of_words = ["this", "is", "a", "list", "of", "words"] items = [word[0] for word in list_of_words] print (items). Code ends.* [Video description ends]

In this case, we will iterate over a list of words. And then create a new list, which contains the first letter in each of the words in our list of words. So the common pattern here is, we need to perform an iteration over each element within a list or a sequence. And then following that, we need to perform some kind of transformation or operation based on those elements. In our case, the operation involves extracting the first character within each element of the list of words.

So here in our list comprehension, we initialize a list within square brackets. And then we set up an iteration over the list of words using this for loop. And for each word within list of words, we extract the character at index 0. This list which we have just created will be assigned to the variable items, which we will then print out. And we can see that this list comprehension works exactly as expected.

[Video description begins] *The output reads: ['t', 'i', 'a', 'l', 'o', 'w'].* [Video description ends]

Each of the first characters for the words in our list have been extracted, and a new list has been created out of it. We now create a similar list, except this time rather than simply extracting the first character of the word directly, we will transform it into uppercase.

[Video description begins] *He enters the line of code: [word[0].upper() for word in list_of_words].* [Video description ends]

So once again, we make use of the list comprehension syntax and iterate over our list of words, and apply the operation word[0].upper for each of the words. And the output in this case happens to be the same list as we had previously, but with uppercase letters.

[Video description begins] *The output reads: ['T', 'I', 'A', 'L', 'O', 'W'].* [Video description ends]

Continuing with our list of words, we once again perform the same iteration. But this time, rather than extracting the first letters in each of the words, we just convert the entire word into uppercase and then create a list out of that.

[Video description begins] *He enters the line of code: [word.upper() for word in list_of_words].* [Video description ends]

So the transformation which we wish to apply in this case is word.upper, which we include just before the for keyword. And then we run the cell and the uppercase versions of each of the words in list of words is now available in this new list.

[Video description begins] *The output reads: ['THIS', 'IS', 'A', 'LIST', 'OF', 'WORDS'].* [Video description ends]

So we have seen that we can create a list by iterating over the elements of another list and then applying a simple transformation to each of those elements. And all of this can be defined within a single line. And this is the true part of list comprehensions. However, what if you do want to iterate over each of the elements within a list, but you don't want your result and list to be of exactly the same size? For example, you wish to perform a transformation on the elements in the list you're iterating over only if they fulfill a certain condition. When I say the words, if they fulfill a certain condition, you should automatically think about an if statement. So is it possible to include if statements within a list comprehension? Well, the answer is yes. So let us create one more list.

[Video description begins] *He enters the line of code: [x * x for x in num if x % 2 != 0].* [Video description ends]

And just as we did in the first example in this demo, we will iterate over the range of integers in the variable num. Now, rather than create a list containing the squares of each of those integers, we want our new list to contain the squares of only the odd integers in num. So the syntax which you use for this is similar to what we saw previously. So within square brackets, we define that a list should contain x times x for every x in num. And then following this, we define the condition for our if statement. So in the list we are creating, it will contain the value x square for every x in the variable num only if x is not divisible by 2. That is if x is an odd number. And when we run this, our resultant list contains the squares of only those integers between 0 and 9 which are odd numbers.

[Video description begins] *The output reads: [1, 9, 25, 49, 81].* [Video description ends]

So let us analyze what has happened here. So if you disregard the transformation of x times x within our syntax here, we have a for loop where we iterate over every x in num. However, the transformation of x times x is not applied for every single element within num. But only for those elements where x mod 2 does not equal 0. So we can make use of list comprehension in order to iterate over every element in a list. But rather than operating on every single element in the list we are iterating over, we can filter its contents by making use of an if statement. Similarly, if we wish to create a new list containing all of the even numbers between 0 and 19, we can also make use of a list comprehension where we apply an if condition.

[Video description begins] *He enters the lines of code, code starts: number_list = [number for number in range(20) if number % 2 == 0] print(number_list). Code ends.* [Video description ends]

Here, we iterate over all of the integers between 0 and 19. And only if the number is divisible by 2, do we include it in our new list? And a list of even numbers is now available.

[Video description begins] *The output reads: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18].* [Video description ends]

Applying Conditions in Comprehensions

[Video description begins] *Topic title: Applying Conditions in Comprehensions. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we saw how we can create new lists out of existing lists by using list comprehensions.

[Video description begins] *The Comprehensions page is open in the jupyter notebook.* [Video description ends]

We also covered how we can include if statements within list comprehensions in order to apply a filter on the list we are iterating over. We can take a look at more such examples. First of all, if we wish to find all of the numbers in the range 0 to 300, which are divisible by the number 27 and then create a new list out of those multiples of 27, we can make use of list comprehensions. So here we iterate over all of the values in the range 0 to 300.

[Video description begins] *He enters the lines of code, code starts: multiples_27 = [i for i in range(300) if i % 27 == 0] print(multiples_27). Code ends.* [Video description ends]

And then we check if the number is divisible by 27. And if it is, we include that number directly in our new list. That is, we do not perform any transformations. So when we run this, all our multiples of 27 are available here.

[Video description begins] *The output reads: [0, 27, 54, 81, 108, 135, 162, 189, 216, 243, 270, 297].* [Video description ends]

Moving along now, if you wish to extract all of the letters in the word mathematics, which are vowels, then we iterate over each of the letters in the word. And then if the value of the letter happens to be either A, E, I, O, or U, then we include it in our new list.

[Video description begins] *He enters the line of code: [i for i in "MATHEMATICS" if i in ["A", "E", "I", "O", "U"]].* [Video description ends]

Again, no transformation is applied on the letters. And this new list contains two A's, an E, and an I.

[Video description begins] *The output reads: ['A', 'E', 'A', 'I'].* [Video description ends]

Now, we will iterate over an entire sentence, and then we will include all of the digits in that sentence in a new list.

[Video description begins] *He enters the lines of code, code starts: string = "Hello 12345 World" numbers = [int(x) for x in string if x.isdigit()] print (numbers). Code ends.* [Video description ends]

So here, the string we are iterating over is Hello 12345 World. And then, in order to iterate over each of the characters, we set up a for loop. Since our new list should only contain the digits within the sentence, we perform a check on each character to verify whether it is a digit for which we call the isdigit function. And if it is, we transform that character into an integer by casting it as an int, and then this integer will be included in our new list. So running this, we'll extract the numbers 1, 2, 3, 4, and 5 which are now separate elements in our new list.

[Video description begins] *The output reads: [1, 2, 3, 4, 5].* [Video description ends]

Similarly, if we wish to extract all of the individual alphabet in that string, we will iterate through the same string as we did previously.

[Video description begins] *He enters the lines of code, code starts: alpha = [x for x in string if x.isalpha()] print (alpha). Code ends.* [Video description ends]

Except this time, we will perform a check on each character to check whether it is an alphabet using the isalpha function. And if it is, that alphabet is included as is in our new list. So running this, all of the individual letters are in a new list.

[Video description begins] *The output reads: ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd'].* [Video description ends]

So it is now becoming clear that it is possible for us to include for loops which include if conditions within our list comprehensions. But what about more complex for loops? For example, is it possible to have nested for loops here? We can go ahead and try it out, where we will perform a similar exercise to what we did previously.

[Video description begins] *He enters the lines of code, code starts: stationery = ["Pen", "Marker", "Ink"] colors = ["Red", "Blue", "Green"] combined = [(i, j) for j in stationery for i in colors] combined. Code ends.* [Video description ends]

Where we iterate over two different lists, a collection of stationary items, and a list of colors, and we produce all possible combinations from those two lists. So in our list comprehension, we iterate over every item in the stationary list. And for each of those items, we iterate over each of the three colors. And at this point, a specific combination of stationary item and color will be available in the variables i and j. What we wish to include in our new list is a tuple which contains that combination of i and j. So this is the transformation which we include in our list comprehension. So when we run this, does it work? And yes, all of those nine combinations are now available as tuples within our list.

[Video description begins] *The output reads: [('Red', 'Pen'), ('Blue', 'Pen'), ('Green', 'Pen'), ('Red', 'Marker'), ('Blue', 'Marker'), ('Green', 'Marker'), ('Red', 'Ink'), ('Blue', 'Ink'), ('Green', 'Ink')].* [Video description ends]

So even within a list comprehension, it is possible for us to make our for loops arbitrarily complex. As long as it is possible for the reader to understand what is going on, this is a very elegant way in which to define a new list. In our next example, we will see how we can include a nested if condition within our list comprehension.

[Video description begins] *He enters the lines of code, code starts: number_list = [number for number in range(51) if number % 2 == 0 if number % 5 == 0] print number_list. Code ends.* [Video description ends]

So here, we iterate over all of the integers from 0 through 50 and we check whether the number happens to be divisible by both 2 and 5. We do this by means of a nested if statement, where we first check if the number is divisible by 2, and if it is, we perform another check to see if it is divisible by 5. So this is just to demonstrate that we can include nested if statements within our list comprehensions. However, you could also implement the same operation by performing a divisibility check for 2 and 5 using an and statement within a single if clause. For this example though, we can just execute this cell and confirm that we can include nested if statements within our list comprehension.

[Video description begins] *The output reads: [0, 10, 20, 30, 40, 50].* [Video description ends]

Now, we don't need to include an if statement just within the for loop of a list comprehension. In fact, we can also use if statements when defining the transformation which needs to be applied for each element, which we iterate over in the for loop.

[Video description begins] *He enters the lines of code, code starts: num = ["Even" if i % 2 == 0 else "Odd" for i in range(10)] print(num). Code ends.* [Video description ends]

So in this case, we iterate over the integers 0 through 9 within a for loop and the value of the integer in each iteration is available in the variable i. In our transformation, we checked the value of i. If it is divisible by 2, then the string Even will be added to our new list, otherwise, the string odd is added. And when we run this, as expected, the strings even and odd appear in alternate positions within the list.

[Video description begins] *The output reads: ['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd'].* [Video description ends]

We continue now with the use of an if condition when performing the transformation for our new list. So now we will iterate over this list of numbers.

[Video description begins] *He enters the lines of code, code starts: numbers = [22, 30, 45, 50, 18, 69, 43, 44, 12] [x+1 if x>= 45 else x + 5 for x in numbers]. Code ends.* [Video description ends]

And our new list will contain x + 1 for every element in our numbers list, which is greater than or equal to 45. And for all of the other numbers, it will include x + 5. And once again, the use of the if statement in the transformation does work.

[Video description begins] *The output reads: [27, 35, 46, 51, 23, 70, 48, 49, 17].* [Video description ends]

We will now take a look at an example, where we make use of if statements, not only in the transformation, but also within our for loop, in order to filter out certain elements in the sequence we are iterating over.

[Video description begins] *He enters the lines of code, code starts: z = ["small" if number < 20 else "large" for number in number if number % 2 == 0 if number % 3 == 0] print(z). Code ends.* [Video description ends]

So we, once again, iterate over the same list of numbers, and then within our for loop, we check whether the number is divisible by both 2 and 3. Effectively whether it is divisible by 6. So the use of this nested if over here, will ensure that the only elements in the numbers list which is in consideration for our new list are those elements which are divisible by 6. And following that, we will apply a transformation on the number which fulfill that condition. So if the number happens to be less than 20, we will include the text small in our new list and will otherwise add the text large. So we can now run this cell.

[Video description begins] *The output reads: ['large', 'small', 'small'].* [Video description ends]

And for the number 30, the text large has been added to a new list. And for the numbers 18 and 12, the text small was included. So list comprehensions are a very powerful and very useful feature in Python. This allows us to very quickly set up a list out of the contents of an existing list, and it is easy for any reader of code to understand what is going on.

Another important thing to keep in mind is that while in this demo, we have only created lists using this list comprehension syntax. If we simply replace the square brackets with parentheses, we will have created a tuple containing the same sequence of elements. It is also possible for us to create sets and dictionaries with the same syntax, which is why they are generally called comprehensions.

Exercise: Advanced Operations in for Loops

[Video description begins] *Topic title: Exercise: Advanced Operations in for Loops. The presenter is Kishan Iyer.* [Video description ends]

In this exercise, you will be writing some code in order to implement some for loops. The first for loop which you will write will iterate over a random list of numbers and then will print those numbers except if the number happens to be divisible by 7. And that is not all, because there is one more condition which you need to incorporate in your for loop, specifically that the iterations will stop if the number 315 is encountered within the list you are iterating over. So there are different courses of action which you can take depending upon the value which you are iterating over in the list. You will need to make use of a couple of the statements which we encountered during this course.

For the next task in the exercise, you will once again start with a random list of numbers. Using that list, you will create another list using list comprehension. And within the new list, you will include the cubes of the numbers from the original list but only if that number happens to be divisible by 3. So just to clarify, you will need to evaluate which of the numbers in your original list is divisible by 3, and you will need to include the cubes of those numbers in your new list. And the creation of the new list will occur in a single line using list comprehension. All of the skills which you need in order to solve each of these problems were covered during this course.

So please pause this video and then spend a few minutes to write the for loops required for this exercise on your own. For the first task in the exercise, you needed to iterate over a given list and then print out all of the values which are not divisible by 7, and also stop the iterations once you encounter the number 315. To incorporate all of those conditions, you need to make use of the break and the continue statements. So let us see how this works. We start off with a list of numbers which include multiples of 7 and also the number 315. So this is something we can use in order to test out our for loop. So let us start iterating over each of the elements within this list_of_nums with the for loop.

Since we will terminate this loop when the number 315 is encountered, we need to perform a check for that number first. So during our iterations, if you find that the value of i is equal to 315, then we stop the iterations by breaking from the loop. Note that since the number 315 is divisible by 7, we will not have to print out its value. In fact, the next thing we do within our for loop is to check whether i is divisible by 7. And if it is, we don't really need to do anything and can just head over to the next iteration in the for loop. So we invoke the continue statement. And then finally, we get to a point where the number is neither 315 and nor is it divisible by 7, in which case, we print out the number.

So once you run this code, you'll get an output similar to this. The numbers 238 and 301 are skipped because they're divisible by 7. And then once the number 315 is encountered, we break out of the for loop, which is why 361 and 399 are never evaluated. Moving along now to the last task in the exercise, which was to create a list containing all of the cubes for the integers in another list which are divisible by 3. So let us consider that our original_list contains all of these elements, the elements 3, 6, 12, and 18 are all multiples of 3.

Now, let us create a cubes_list where we iterate over all of the elements in the original_list and check whether the element is divisible by 3. If it is, then the cube of that element will be included in our cubes_list. So within this single line, we have defined what our list of cubes will look like. And to confirm your logic, you can print out the contents of cubes_list, and it should look something like this. Where the cubes of 3, 6, 12, and 18 will be included.