Conditional Statements & Loops: The Basics of for Loops in Python

Loops are one way to perform the same operations repeatedly in a program. For loops are the control structure to use when the repeated operations are performed on a sequence such as a list or a tuple. In this 9-video course, you will explore different ways to iterate over a sequence using for loops. Key concepts covered in this course include how to use for loops to process elements in a list and characters in a string; and how to code for loops to iterate over values in a tuple and the keys and values in a dictionary. Next, learn the function of associating an else block with a Python for loop; include if-else statements and other for loops within a for loop; how to generate a sequence of consecutive integers with the range function; and how to use the range function to iterate over a large range of values and apply it within nested for loops. Finally, observe how to write for loops in order to iterate over 1-dimensional and 2-dimensional sequences.

Table of Contents

Course Overview

[Video description begins] *Topic title: Course Overview. Your 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, The Basics of 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 Masters 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 is the default sequential control structure, in which statements execute one after another.

The second, our conditioner control structures in which the 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 terminating condition evaluates to true. In this lab only course, we explore one of the most basic iterative control structures, namely the for loop. We begin by exploring how for loops can be used to process the elements of a list one by one, and perform repetitive actions with those elements. We then cover how for loops work with any data which represents a sequence in Python, such as tuples, sets, and even dictionaries. We go on to adding more complexity to our for loops by introducing if statements to evaluate conditions and even including nested for loops.

That is for loops within for loops. We then explore the use of the built-in range function, in order to iterate over a specific sequence of integers. Once you do all of the labs in this course, you will have built a firm foundation for

the use of for loops. You can then move on to more advanced topics related to for loops. Such as controlling how to end such a loop, and then skipping actions for specific iterations.

Iterating over Elements in a List

[Video description begins] *Topic title: Iterating over Elements in a List. The presenter is Kishan Iyer.* [Video description ends]

In this course on the use of for loops in Python, we will be making use of Jupyter notebooks as our IDE. Jupyter notebooks are a web-based and interactive IDE

[Video description begins] *The jupyter notebook opens in the web browser. It is divided into three parts. The first part is a menu bar. It includes Files and Running tabs. The second part includes a New drop-down button. The third part is a content pane. The Files tab is selected and the corresponding page opens in the content pane. The following text is displayed on the page: The notebook list is empty.* [Video description ends]

which is very popular among Python developers, as well as data scientists in general. Its interactive interface allows programmers to iteratively develop their applications by executing just parts of their code. And then validate their code before moving along and expanding on it. I have now brought up the Jupyter notebooks application in my own web browser. And as you can see, my workspace is currently empty.

[Video description begins] *He highlights the following text: The notebook list is empty.* [Video description ends]

So let us go ahead and create our first Jupyter notebook in order to implement for loops. For that, I'm going to head over to this New button and then select the Python 3 kernel.

[Video description begins] *He clicks the New drop-down button and the drop-down list opens. It includes Python 3 and Terminal options.* [Video description ends]

With that done, a new notebook has popped up, but this has been given the default name of Untitled.

[Video description begins] *The Untitled page opens in the jupyter notebook. It is divided into three parts. The first part includes File and Edit options. The second part includes a Run button. The third part is a content pane and it contains multiple cells.* [Video description ends]

Let us go ahead and change that by clicking on this. And then for this demo, I'm going to call the notebook IntroducingForLoops.

[Video description begins] *He clicks the title: Untitled and the Rename Notebook dialog box opens. It contains Enter a new notebook name text box and Cancel and Rename buttons.* [Video description ends]

All right, with the notebook setup, it is not time for us to begin coding.

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

So in the first cell of the notebook, I will just go ahead and paste the first for loop which we will implement. The first question to ask at this point, though is, what exactly is a for loop? One way to put it is that a for loop allows us to process the elements in a sequence one at a time. That is, it allows us to iterate over the contents of that sequence. And in the context of Python, you will know that sequences could be in the forms of list, tuples, dictionaries, sets, and so on. All right, so what is the first sequence we will iterate over? So this is going to be a list which contains the letters of the word, Hello.

[Video description begins] *He enters the lines of code, code starts: for letter in ['H', 'e', 'l', 'l', 'o']: print(letter). Code ends.* [Video description ends]

So we will now iterate over each of these characters and then print them out. So we will be going through the characters H, e, l, l, and o, one at a time, and each time we process an alphabet we will refer to that as an iteration. So how exactly is this sequence of iterations defined? Well, for that we create this for block, which begins with the keyword for in the first line.

[Video description begins] *He highlights the line of code: for letter in ['H', 'e', 'l', 'l', 'o']:*. [Video description ends]

And that line ends with the sequence over which we need to iterate, which in our case, is a list of characters making up the word Hello. So what exactly happens in each iteration when we setup a for loop? Well, a particular character from the list starting from the first one will be assigned to this variable letter. And within the body of the for block, we can decide exactly what needs to be done at each iteration.

So in our example, because there are five characters which need to be processed, the contents of the for block will run exactly five times. Let us now run this cell in order to see what the output will look like. If you're still unfamiliar with Jupyter notebooks, the way to run a cell is to select it. And you could either use the Run button at the top or you could use a keyboard shortcut of either Shift + Enter, or Ctrl + Enter in the case of a macOS.

[Video description begins] *The output reads: H e l l o*. [Video description ends]

So on running this cell, we can see that each of the letters of the word Hello have been printed on a new line. This is because in each iteration of our for loop, the variable letter was assigned the characters of our list in sequence. So in the first iteration, the value of letter was upper case H, and then it printed out that value. It then moved on to the next iteration where the value of letter was e, and then that was printed out. For the third and fourth iterations, the value of letter was l, and then it was o in the last of these iterations.

So the for loop has allowed us to process all of the elements of this sequence of characters one at a time. And it allows us to assign the value of an element in that sequence to a variable. Now let us perform a slight variation of the for loop which we just executed.

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

Now rather than explicitly define a list containing the characters in the word Hello, we simply asked the for loop to iterate over a string Hello. And what exactly is the output? Well, it is exactly the same as our previous execution.

[Video description begins] *The output reads: H e l l o*. [Video description ends]

But why exactly does this happen? Well, when we start off our definition of a for loop, we make use of the for keyword and that line ends with a sequence over which we wish to iterate. While a list is very obviously a sequence, a string is not. But Python is smart enough to assume that if you're iterating over a string, what we really mean is that we wish to iterate over the characters which make up that string one at a time.

Which means that these two cells which we have executed are pretty much exactly the same. Because the string, Hello, has been converted into a list containing the characters, H, e, l, l, and o in that order. In fact, we can iterate over a string for a second time, except on this occasion, we will assign that string to a variable called my_string.

[Video description begins] *He enters the lines of code, code starts: my_string = "Hello World" for char in my_string: print (char). Code ends*. [Video description ends]

And then this time, at each iteration, a character in our string will be assigned to the variable char. And within each iteration, we will simply print out the contents of that char variable. So we can once again, run this cell. And in the output, we see each of the characters which make up the string,

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

Hello World. You will notice that even the space has been treated as a separate character which is why you see what looks like a blank line between the o and W characters. However, that line is not quite blank but contains a space. We will now continue iterating over the characters in a string. So in this occasion, we use the number 10, which is expressed in the form of a string.

[Video description begins] *He enters the lines of code, code starts: for digit in "10": print (digit). Code ends*. [Video description ends]

So note that we have specified 10 within quotes here, which is why it is a string and not a number. So what happens when we execute this for loop? So on running the cell, we see the digits 1 and 0 which are printed out in each iteration.

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

This is because even if we know that this represents a number, in the case of Python, it is still a string which contains two characters. And when this string is used in a for loop, it is effectively treated as a list or a sequence of two characters. Now what if instead of using a string to represent the number 10 we actually used a numeric value?

[Video description begins] *He enters the lines of code, code starts: for digit in 10: print (digit). Code ends*. [Video description ends]

So this is an integer 10 which we are planning to iterate over. Since we are not enclosing 10 within quotes, this is not a string and will be an integer. So can we use it in a for loop in this manner? And the answer is no. The error message which we see here tells us that the int object is not iterable. So the object which we need to iterate over in a for loop needs to be of an iterable nature. So in the world of Python, lists, tuples, dictionaries and so on, and also strings, as we have just covered, are iterable objects. But this is not the case with integers, floating point numbers, boolean values, and many other types. So when we do use a for loop, we need to ensure that the object which we're planning to iterate over is in fact something which is iterable in the context of Python.

Iterating over Elements in a Tuple and Dictionary

[Video description begins] *Topic title: Iterating over Elements in a Tuple and Dictionary. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we took a look at how we can iterate over a number of elements in a string using a for loop and how we can also iterate over the characters in a string.

[Video description begins] *The IntroducingForLoops page opens in the jupyter notebook*. [Video description ends]

What if, though, we don't use a list or a string, but in fact a collection of comma separated strings? So here we have strings representing three different countries which are separated by commas.

[Video description begins] *He enters the lines of code, code starts: for country in "Germany", "India", "Israel": print(country). Code ends*. [Video description ends]

However, they are not within square brackets in which case they would form a list. So the questions which come to mind at this point are, will this throw an error? Will the for loop iterate over each of the individual characters in these strings? Or will this be treated as a list of three strings? And when we run this cell, we get the answer that it is the last of those.

[Video description begins] *The output reads: Germany India Israel*. [Video description ends]

So when we have values which are separated by commas, even if they're not placed within square brackets, Python is smart enough to assume that this is to be treated as a sequence where the elements are separated by commas. So the elements we have iterated over in this case are the strings representing the countries.

[Video description begins] *He enters the line of code: dogs = ("Pug", "Doberman", "Golden Retriever").* [Video description ends]

We move along now and then specify a tuple containing three different elements over which we can iterate. So this is a tuple representing three breeds of dogs. And each of these breeds is represented with a string. So what happens if we were to use this tuple in a for loop?

[Video description begins] *He enters the lines of code, code starts: for dog in dogs: print("It's a", dog). Code ends.* [Video description ends]

We assume that the element in each iteration will be a string which represents the dog breed.

[Video description begins] *He highlights the line of code: print("It's a", dog).* [Video description ends]

And on this occasion, we won't just print out the value directly, but in each iteration we will print out the string, it's a, followed by the element which we expect to be the string for the dog breed. And when we run this cell, we see that our expectation is spot on.

[Video description begins] *The output reads: It's a Pug It's a Doberman It's a Golden Retriever.* [Video description ends]

So here we have in fact constructed a full sentence using the element in a tuple at each iteration. So the important takeaway here is that in each iteration of a for loop, we will have access to an element over the sequence we are iterating over.

[Video description begins] *He highlights the line of code: for dog in dogs:.* [Video description ends]

And it is up to us what exactly we do with that element. Whereas earlier on we started off by simply printing out the value of the element as is. Here we have used that element in order to form a sentence.

[Video description begins] *He highlights the line of code: print("It's a", dog).* [Video description ends]

Let us move along now and iterate over some more complex data types.

[Video description begins] *He enters the lines of code, code starts: dog_weight = (("Pug", 20), ("Doberman", 80), ("Golden Retriever", 55)) for i, (dog, weight) in enumerate(dog_weight): print("The dog at index %d is a %s and it weighs %s pounds" % (i, dog, weight)). Code ends.* [Video description ends]

Whereas previously, the element which we had access to in each iteration was either a character or a string, when we are iterating over a tuple of tuples, the element at each iteration will be an individual tuple. So in this case, we have a tuple of three elements, each element being a tuple. And this individual tuple represents the name of a dog breed, which is a string. Along with the average weight for that dog breed in pounds, which is an integer.

So here we assign this tuple of tuples to this variable dog_weight. And then we iterate over this tuple of tuples using a for loop. However, do note here that we're not iterating over the dog_weight variable directly, but we are using an enumerate function. What exactly does this function do? Well, we will see that it will return at each iteration not only the individual element, which is a tuple, but also an index for that element within the object which we are iterating over.

[Video description begins] *He highlights the line of code, code starts: print("The dog at index %d is a %s and it weighs %s pounds" % (i, dog, weight)). Code ends.* [Video description ends]

So at each iteration, we will expect two different values. The variable i will contain the index for the element. And then the element itself will be a tuple containing the dog breed and its weight. We are not capturing the element within a single variable, but in fact we are forming a tuple out of it. And the elements of that tuple will be available within the variables dog and weight at each iteration. Within the for loop body we define what to do at each iteration, which is to access these values and then use them in a sentence. So we just run this and we can see exactly how this sentence is formed.

[Video description begins] *The output reads: The dog at index 0 is a Pug and it weighs 20 pounds The dog at index 1 is a Doberman and it weighs 80 pounds The dog at index 2 is a Golden Retriever and it weighs 55 pounds*. [Video description ends]

At each iteration, the variable i contains the index. The variable dog contains the first element within the tuple. And then the weight variable contains the second of those elements. All right, let us go ahead and then implement something a little simpler this time.

[Video description begins] *He enters the lines of code, code starts: friends = ["John", "Sam", "Jill"] for friend in friends: print("Happy New Year,", friend). Code ends*. [Video description ends]

So let's say we have a list of three different friends. And we wish to iterate over this list and then print out a Happy New Year message to each of them. So here we iterate over a list of strings. And when we run this, we can confirm that the element which is available at each iteration, that is the value of the variable friend, happens to be one of the strings in our list of friends.

[Video description begins] *The output reads: Happy New Year, John Happy New Year, Sam Happy New Year, Jill*. [Video description ends]

So we have now covered what needs to be done in order to iterate over a list and also a tuple, but what about a Python dictionary? How exactly can we iterate over these?

[Video description begins] *He enters the lines of code, code starts: student_scores = {'John': 80, 'Sam': 60, 'Jill': 50, 'Bob': 96} student_scores. Code ends*. [Video description ends]

So assume that we have a dictionary containing four key value pairs. The keys represent the names of certain students and the corresponding values or their scores in some course. So now our student scores dictionary contains these four elements.

[Video description begins] *The output reads: {'John': 80, 'Sam': 60, 'Jill': 50, 'Bob': 96}*. [Video description ends]

But now, what if we were to iterate over the contents? So if we were to use this student_scores dictionary in a for loop, what is the value of student at each iteration?

[Video description begins] *He enters the lines of code, code starts: for student in student_scores: print(student). Code ends*. [Video description ends]

Is it the key, is it the value, or is it a combination of the key and the value? So let us just run this. And what the output suggests is that our for loop seems to iterate over the keys in our dictionary.

[Video description begins] *The output reads: John Sam Jill Bob*. [Video description ends]

As you are aware, within a dictionary, the order of the elements should not matter here. So the order of the keys should not be a factor. However, what if we were to create a list out of the dictionary's keys for which we can make use of a dictionary's keys function.

[Video description begins] *He enters the line of code: student_scores.keys()*. [Video description ends]

So here we have a list of all these keys.

[Video description begins] *The output reads: dict_keys(['John', 'Sam', 'Jill', 'Bob'])*. [Video description ends]

And if we were to iterate over this list, we can see that the output is exactly the same as iterating over the dictionary directly.

[Video description begins] *He enters the lines of code, code starts: for student in student_scores.keys(): print(student). Code ends*. [Video description ends]

So we can see that by default when we iterate over the contents of a dictionary, we are in fact iterating over its keys.

[Video description begins] *The output reads: John Sam Jill Bob*. [Video description ends]

Now, what if we wanted to go over the values in a dictionary one by one rather than the keys?

[Video description begins] *He enters the lines of code, code starts: for score in student_scores.values(): print(score). Code ends*. [Video description ends]

For that, we can make use of the dictionary's value function, which will create a list of values.

[Video description begins] *The output reads: 80 60 50 96*. [Video description ends]

And when we iterate over that, all of the scores of the students are what are accessible at each iteration. So we now know how we can iterate over the keys as well as the values individually using a for loop. However, if we wanted to access both the key and the corresponding value, well, since the value is accessible from the key, we can just iterate over the keys themselves.

[Video description begins] *He enters the lines of code, code starts: for key in student_scores: print("Key value pair -", key, ":", student_scores[key]). Code ends*. [Video description ends]

And then use the key in order to access the corresponding value by referencing the student_scores dictionary. So when we run this, both the key as well as the value is accessible for the dictionary.

[Video description begins] *The output reads: Key value pair - John : 80 Key value pair - Sam : 60 Key value pair - Jill : 50 Key value pair - Bob : 96*. [Video description ends]

However, there is a simpler way to do this. And this is thanks to a function called items which is available for a dictionary object in Python.

[Video description begins] *He enters the lines of code, code starts: for key_value in student_scores.items(): print (key_value). Code ends*. [Video description ends]

So this will return a tuple at each iteration, where the tuple contains the key as well as the value. So here we iterate over each element in student_ scores.items. And we see that the elements here are in fact the key and value pairs in the form of a tuple.

[Video description begins] *The output reads: ('John', 80) ('Sam', 60) ('Jill', 50) ('Bob', 96)*. [Video description ends]

Now, just like we did previously, let us now break down this tuple into its elements since we know that each tuple in student_scores.items contains a key and a value pair.

[Video description begins] *He enters the lines of code, code starts: for student, score in student_scores.items(): print ("Student:", student, "\tScore:", score). Code ends*. [Video description ends]

So here we will capture the key in the variable student, and the variable score will contain the corresponding value. So now that we have the key and values in separate variables, we can use them individually in order to construct a sentence at each iteration. So we run this cell, and this is the output which is generated.

[Video description begins] *The output reads: Student: John Score: 80 Student: Sam Score: 60 Student: Jill Score: 50 Student: Bob Score: 96*. [Video description ends]

The else Block of a for Loop

[Video description begins] *Topic title: The else Block of a for Loop. The presenter is Kishan Iyer.* [Video description ends]

So we have now used the elements which are available to us at each iteration in order to either just print out the elements directly or to form sentences.

[Video description begins] *The IntroducingForLoops page opens in the jupyter notebook*. [Video description ends]

But now, we shall use them in order to perform some arithmetic operations. So in this case, we will now be iterating over a list of numbers.

[Video description begins] *He enters the lines of code, code starts: numbers = [4, 6, 9, 10] for num in numbers: quotient = num // 2 print(quotient, "is the quotient of", num, "/ 2"). Code ends*. [Video description ends]

And in each iteration, we will divide the number by 2 and then print out the quotient of that division. So to do that, we initialize this list of numbers. And then in our for loop, we will iterate over each of the numbers within the numbers list. And then we will use that in order to generate a quotient value by dividing the value of num by 2. So note here that we specify this double slash operator which is used to return the quotient. And once we have the quotient, we will use it to form a sentence. So let us run this.

[Video description begins] *The output reads: 2 is the quotient of 4 / 2 3 is the quotient of 6 / 2 4 is the quotient of 9 / 2 5 is the quotient of 10 / 2*. [Video description ends]

And you can see here that the quotient is available at each iteration. Once again, I would like to stress that the goal of a for loop is to allow us to iterate through the individual elements in a sequence. So at each iteration, this individual element will be available to us. And if the task of processing each of those elements is something which is repetitive, then we can write the code for that repetitive process within the body of the for loop. The process which we do define within the for loop body can be very simple, could also be very complicated, or could be somewhere in between. Let us move along now, and then on this occasion, we will be iterating over a list once more.

[Video description begins] *He enters the lines of code, code starts: mixed_list = [145, 10.5, 1+3j, True, "Python", (0, 1), [2, -5], {"Class":"V", "Section":"A"}] for item in mixed_list: print ("Type of ", item, " is ", type(item)). Code ends*. [Video description ends]

But unlike the previous lists which we have gone over, this one contains mixed data. That is, the types are not consistent. You can already see here that the number 145 is an integer. The second element, which is 10.5, is a floating point value. The third element in our list is a complex number. The fourth one is a boolean value followed by a string. We have a tuple, we have a list, and then finally a dictionary. We then set up a for loop to iterate over this mixed list and then print out, at each iteration, the element itself, along with its type. For that, we make use of the Python type function. So what exactly happens when we iterate over this melting pot of a list? And the answer is that the for loop works.

[Video description begins] *The output reads: Type of 145 is <class 'int'> Type of 10.5 is <class 'float'> Type of (1+3j) is <class 'complex'> Type of True is <class 'bool'> Type of Python is <class 'str'> Type of (0, 1) is <class 'tuple'> Type of [2, -5] is <class 'list'> Type of {'Class': 'V' , 'Section': 'A'} is <class 'dict'>. Code ends*. [Video description ends]

We are indeed able to access each of the elements. So it is clear that it is possible for us to iterate over a sequence, even if the sequence does not contain data of the same type. It is now time for us to do some problem solving. So the first of the problems is to iterate over all the numbers in a list of integers and then print out the squares of those numbers.

[Video description begins] *He enters the lines of code, code starts: num_list = [2, 4, 6, 8] square = 0 for val in num_list: square = val ** 2 print ("Square of", val, "is", square). Code ends*. [Video description ends]

Since we need to access the individual elements in the list one by one, the for loop is the natural solution for us. So we setup this for loop. And within each iteration of the for loop, and within the body of the for loop, we need to define exactly what needs to happen with each element. Since our problem is to calculate the squares of each of the elements, we will perform that calculation by applying the power of 2 for each of the elements in our list. And then we will print out that value at each iteration as well.

[Video description begins] *The output reads: Square of 2 is 4 Square of 4 is 16 Square of 6 is 36 Square of 8 is 64*. [Video description ends]

So here, we run this, and the squares of each of the four elements within our list are available in the output. For the next problem, we will accept a string from the user as an input and then calculate the length of that string without using the len function available within Python.

[Video description begins] *He enters the lines of code, code starts: string = input("Enter a string: ") count = 0 for i in string: count = count + 1 print(count). Code ends*. [Video description ends]

How exactly can we do that? Well, one possibility is to count the number of characters. So, if we want to break down a string into a number of characters and then count them one by one, well, once again, it is a for loop which is the natural solution. So here, we initialize a variable called count to 0. And then we iterate over each of the individual characters within the string by using the for loop. And then, within each iteration, we don't even access the characters, but we simply increment the value of count by 1. At the end of the for loop, we will print out the value of count, and this should give us the final answer for the string length. So let us run this cell.

[Video description begins] *The output Enter a string: is displayed*. [Video description ends]

When prompted for an input, I will enter the string python and hit Enter. And immediately, we are returned a value of 6, which is the number of characters in python. All right, so with those problems solved, we can move along with for loops and take a look at one of its additional features, which is available in Python but not in many other programming languages. And this is the use of an else statement along with a for loop.

[Video description begins] *He enters the lines of code, code starts: numbers = [2, 3, 5, 7] for num in numbers: print(num) else: print("No more items left in the list."). Code ends*. [Video description ends]

So here, we are implementing a for loop which iterates over a list of integers. And then after that, we also have an else block within which we simply print out the message, No more items left in the list. So when an else block is used along with a for loop, it means that the execution of the code will enter the else block after all of the iterations of the for loop have been completed successfully. So when we run this, we can see that the for loop has completed four iterations, as we expect it to.

[Video description begins] *The output reads: 2 3 5 7 No more items left in the list*. [Video description ends]

And since all the four iterations were completed without any issues, the else block is also executed. So when is the else block not executed when used with the for loop in this manner? Well, we could either have an error during the processing of each element within the for loop, or we could also exit prematurely from the for loop before it gets to complete all of the iterations. Now that is something which we will cover just a little later in this course. So we can now perform a similar iteration, but this time over a set of characters.

[Video description begins] *He enters the lines of code, code starts: characters = {"a", "b", "c", "d"} for letter in characters: print(letter) else: print("No more characters left "). Code ends*. [Video description ends]

And we once again make use of an else block to print out the message that there are no more characters left for us to iterate over. So can they be issues when we iterate over a set? And the answer is no.

[Video description begins] *The output reads: d b a c No more characters left*. [Video description ends]

We go through four iterations, as we expect to, and then the else block is also executed. One thing to note here is that when we iterate over the set, the order of the elements which have been processed is not exactly an alphabetical order. Which alludes to the fact that when we iterate over the contents of a set data structure, we should not expect a specific order.

Nested Control Structures in a for Loop

[Video description begins] *Topic title: Nested Control Structures in a for Loop. The presenter is Kishan Iyer.* [Video description ends]

It is time to move ahead and then see how we can make use of an if statement within the body of a for loop.

[Video description begins] *The IntroducingForLoops page opens in the jupyter notebook*. [Video description ends]

I assume here that you're already familiar with using an if statement in order to evaluate a condition, but what if you wanted to evaluate a condition for each of the elements within a list or a tuple or some other sequence? One way to phrase this is that you wish to perform a specific set of evaluations repeatedly for every element within a list, and this is precisely where you would use a for loop.

[Video description begins] *He enters the lines of code, code starts: us_cities = ["New York", "Nashville", "Seattle"] for city in us_cities: if city == "New York": print("New York is present in the list"). Code ends*. [Video description ends]

Here we have a list containing three different strings, which represent cities within the United States. We iterate over each of these, and if the city happens to be New York, we will print out the message that New York is present in the list. So here you can see that we iterate over the cities and then we check whether the value of the city in any iteration is equal to New York using an if statement. And then within the body of the if statement, we print out the message that New York is in the list. So let us see whether this works by running the cell.

[Video description begins] *The output reads: New York is present in the list*. [Video description ends]

And yes, this print statement executes exactly once, which matches the single occurrence of New York within our list. Similarly, we can now iterate over a collection of integers and see if there is any even number within that list.

[Video description begins] *He enters the lines of code, code starts: numbers = [11, 33, 55, 39, 2, 55, 75, 37, 22, 23, 41, 13] for num in numbers: if num % 2 == 0: print("The list contains an even number:", num). Code ends*. [Video description ends]

You will observe that this list contains pretty much odd numbers, except for two values, 2 and 22. So to identify the even numbers, we will iterate over the numbers list using a for loop. And then at each iteration, we get the value of the number in the variable num. We then use the if statement in order to check whether that number is divisible by 2 for which we use the modulus operation. And if it is, we print out the message that the list contains an even number. So let us go ahead and run this cell, and what we see is that the print statement executes exactly twice.

[Video description begins] *The output reads: The list contains an even number: 2 The list contains an even number: 22*. [Video description ends]

So we were able to iterate over an entire list, and then use the if statement, on each of the individual elements within that list. Let us now perform a similar operation, but on this occasion we will make use of not just an if block, but also an else block corresponding to the if.

[Video description begins] *He enters the lines of code, code starts: years = [2016, 2017, 2018, 2019, 2020] for year in years: if year % 4 == 0: print(year, "is a leap year") else: print(year, "is not a leap year"). Code ends*. [Video description ends]

Again, note here that this else pairs with an if statement and not with the for loop. So here we have a list of five different years and then we wish to check whether any of those are leap years. We will print out the message that the year is a leap year, if it is one. And then if it is not, we will print out another message that the year is not a leap year. So we wish to perform this if else evaluation for each element within a list which is why it makes sense to place the if else block within a for loop.

[Video description begins] *The output reads: 2016 is a leap year 2017 is not a leap year 2018 is not a leap year 2019 is not a leap year 2020 is a leap year.* [Video description ends]

So we run this for our list of five elements, and we can see that it enters the if block on two occasions for the two leap years, 2016 and 2020. And then on the other three occasions it went into the else block and printed out that those years are not leap years. Let us try something a little different now. So on this occasion, the object which we will iterate over will be a list of lists. So here we have a list of names, but then within that list are two lists.

[Video description begins] *He enters the lines of code, code starts: names = [["Mary", "Linda", "Jessica"], ["James", "Jacob", "William"]] for sublist in names: print(sublist). Code ends*. [Video description ends]

The first list contains a collection of female names, and the second one a collection of male names. So we setup a for loop in order to iterate over each sublist within the broader list and then we will print out the sublist.

[Video description begins] *The output reads: ['Mary', 'Linda', 'Jessica'] ['James', 'Jacob', 'William']*. [Video description ends]

And we can see that the variable sublist in each iteration is itself a list. So the element available to us in each iteration is itself a sequence and is it possible for us to iterate over that? And the answer is yes.

[Video description begins] *He enters the lines of code, code starts: for sublist in names: for name in sublist: print(name). Code ends*. [Video description ends]

So, in this case, we will be iterating over not just the list of names, but also the individual sublists within that, and for that, we will make use of a for loop within a for loop. Such a setup is known as a nested for loop. So let us quickly analyze what we're doing here. So for each sublist with the names, we access the sublist and then we iterate over that and then for each name within that sublist, we just print out the name.

[Video description begins] *The output reads: Mary Linda Jessica James Jacob William*. [Video description ends]

So when we run this, you can see that all of the six names appear on different lines. So it is possible for us to have for loops within for loops and in fact, we can have for loops up to any levels as long as they're able to

manage all the data. So while nested for loops are very useful when we are iterating over, say, a list of lists or similar nested sequences, we can also use nested for loops in order to generate combinations out of two different lists. So here we have a list of colors, and then after that we have a list of objects.

[Video description begins] *He enters the lines of code, code starts: colors_list = ["Red", "Green", "Blue"] objects_list = ["Pen", "Marker", "Pencil"] for color in colors_list: for obj in objects_list: print (color, obj). Code ends*. [Video description ends]

So, if a problem is to generate all possible combinations of objects and colors using these two lists, we can in fact make use of a nested for loop. So we'll start off by iterating over each individual color within the colors list. And then we will pair each individual color with all the possible objects from the object list.

So for example, we will start off with the color red in the colors list. And then we will combine it with each of the objects, pen, marker, and pencil, in order to print out a red pen, a red marker, and a red pencil. We will then do the same thing for green and blue as well. So here, let us just go ahead and run the cell.

[Video description begins] *The output reads: Red Pen Red Marker Red Pencil Green Pen Green Marker Green Pencil Blue Pen Blue Marker Blue Pencil*. [Video description ends]

And you can see that for two lists containing three elements, a total of nine different combinations have been printed out.

An Introduction to the range Function

[Video description begins] *Topic title: An Introduction to the range Function. The presenter is Kishan Iyer.* [Video description ends]

In the previous video, we saw how we can use a for loop in order to iterate over each of the individual elements which are present in some sequence, such as a list, a tuple, and so on. The number of iterations of a for loop in all of those instances was dependent upon the size of the structure over which we were iterating.

[Video description begins] *He opens the RangeFunctionInForLoop page in the jupyter notebook. It is divided into three parts. The first part includes File and Edit options. The second part includes Run button. The third part is a content pane and it contains multiple cells*. [Video description ends]

So if we were iterating over a list containing n elements, then the number of iterations in our for loop will be n. Now what if we would like to explicitly define the number of iterations on our own? For that, there is something known as a range function which is supplied by Python. And in this demo, we will take a look at exactly how that can be used. Now I have created a new Jupyter notebook for this demo, and then I have also renamed it to RangeFunctionInForLoop. So once you have created such a Jupyter notebook, we can begin coding. And the first thing we will do is to create a new range object.

[Video description begins] *He enters the lines of code, code starts: my_range = range(5) my_range. Code ends*. [Video description ends]

So this is a variable called my_range. And we assign it a value, which is returned by a call to a range function. Just in case you are not familiar with the notion of functions yet, just remember for now that it is something which can take in an input, it'll perform some processing. And after that, it may return a value which you can assign to variables. In this case, we are making a call to a function called range. We are supplying it an input of 5, and then it returns a value which we capture in this variable, my_range. So we'll now take a look at what that variable contains for which we will run this cell. And what you see is something known as a range object.

[Video description begins] *The output reads: range(0, 5)*. [Video description ends]

What it actually represents is a sequence of numbers from 0 through 4.

[Video description begins] *He enters the line of code: tuple(my_range)*. [Video description ends]

To confirm that, we can in fact, convert this my_range variable into a tuple, for which we will call the tuple function.

[Video description begins] *The output reads: (0, 1, 2, 3, 4)*. [Video description ends]

And you see that this does return to us a sequence of numbers from 0 through 4.

[Video description begins] *He enters the line of code: list (my_range)*. [Video description ends]

Similarly, if you were to apply the list function in order to transform our my_range variable to a list, it again, returns the numbers 0 through 4.

[Video description begins] *The output reads: (0, 1, 2, 3, 4)*. [Video description ends]

So what exactly did this call to the range function give us? So when we supplied it with an input of 5, it gave us all the values from 0 until one less than 5. In other words, it gave us a range of values. So when we supplied one argument or one input to the range function, it assumed that the start of the range was 0 and that the end of the range was one less than the input.

[Video description begins] *He enters the line of code: list(range(5, 10))*. [Video description ends]

So now, if you were to call the range function, but this time we supply two arguments or two inputs, how exactly would it work? Given that this is a range function, we should expect it to return values within the range 5 and 10, because those are the two arguments to the function. So when we run this and apply the list function, we can see that it does indeed return the values in that range.

[Video description begins] *The output reads: [5, 6, 7, 8, 9]*. [Video description ends]

However, once again, the maximum value in the range is one less than the maximum value which we supplied to the function. So it is now becoming clear that this range function gives to us a sequence of integers within the given range. So what if you were to specify a range which contains negative values?

[Video description begins] *He enters the line of code: list(range(-5, 5))*. [Video description ends]

So in this instance, let's say we would like all of the values in the range -5 to 5. And when we run this, we do indeed get all of those integers within that range.

[Video description begins] *The output reads: [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]*. [Video description ends]

So the range function also works with negative values. Another important thing to note is that since we are able to convert what is returned from the range function into a list or a tuple, it is something which we can iterate over. This means that if you would like to iterate over a range of values, we can in fact use the range function within a for loop. So let us try that out.

[Video description begins] *He enters the lines of code, code starts: for i in range (3.3): print (i). Code ends*. [Video description ends]

However, when we make use of the range function in a for loop, this time, let us specify an argument which is in fact a floating point number. Will that work? Because so far we have only been working with integers. So when we run this, we get an error method saying that the float object cannot be interpreted as an integer. So it does look as though the range function only works with integers and not floats.

[Video description begins] *He enters the lines of code, code starts: for i in range ("C", "Y"): print(i). Code ends*. [Video description ends]

We will now make use of another for loop, and this time we will also make use of the range function, but we will specify a range between two alphabets, C and Y. So does it work with strings?

[Video description begins] *The output displays the TypeError: 'str' object cannot be interpreted as an integer.* [Video description ends]

So when we run this, once again, we get a similar message to what we did previously. And clearly, the range function does not like string arguments as well. So we yet haven't been able to use the range function within a for loop, so it's not clear yet whether that works. So let us go back to something which is more tried and tested.

[Video description begins] *He enters the lines of code, code starts: for i in range(6): print(i). Code ends.* [Video description ends]

So here, we will initialize another for loop, but this time we will use a range function call which we know works. That is, we pass along a single argument, a single input, which is an integer. So if this works, the range function should give us a sequence of integers from 0 through 5, and we will print each of these values within the for loop. And this for loop does work as expected.

[Video description begins] *The output reads: 0 1 2 3 4 5.* [Video description ends]

The range of integers, from 0 through 5, are indeed printed out here. And we also know for sure this time that the range function can be used with a for loop. And this is where this function comes in handy. If we want the for loop to execute exactly six times, then we simply use the range function and then pass it a value of 6. So if you would like the word hello to be printed out a total of 3 times, we will initialize another for loop.

[Video description begins] *He enters the lines of code, code starts: for i in range (3): print ("Hello"). Code ends.* [Video description ends]

And on this occasion, we use the range function and pass it a value of 3, which means that there will be a total of 3 iterations of this for loop. And within the body of the for, we will print out Hello. And on running this, the text Hello is printed out a total of 3 times.

[Video description begins] *The output reads: Hello Hello Hello.* [Video description ends]

So note here that the value of i is entirely irrelevant in this particular loop. All we wanted was for the print statement to execute a set number of times, and it did not really matter to us what the value of i was. So when you do encounter such instances where you have a for loop or some kind of loop and the value of a variable does not really matter in each iteration, then you can simply use the character _ as the name of the variable. So here, we replace i with _, and we run this for loop.

[Video description begins] *He enters the lines of code, code starts: for _ in range(3): print ("Hello"). Code ends.* [Video description ends]

And of course, the output is exactly the same. So this is just a convention, and not just Python, but in other programming languages.

[Video description begins] *The output reads: Hello Hello Hello.* [Video description ends]

When you use _ as the name of the variable for an element in a loop, when the value of that element is not really going to be accessed.

Setting Intervals in a Range

[Video description begins] *Topic title: Setting Intervals in a Range. The presenter is Kishan Iyer.* [Video description ends]

We will continue now with our exploration of the range function.

[Video description begins] *He opens the RangeFunctionInForLoop page in the jupyter notebook. The following lines of code are displayed, code starts: for i in range (2 ** 2): print ("python"). Code ends.* [Video description ends]

So the arguments to such a function, we have seen, need to be integers. But it is not necessary to pass along integers directly to the function. You can in fact even pass along mathematical expressions which evaluate to integers. So in this case, we pass along as argument, 2 to the power 2, which evaluates to 4. And on executing this for loop, we see that the print function is in fact executed four times.

[Video description begins] *The output reads: python python python python.* [Video description ends]

And just to confirm that we can use a minimum as well as a maximum for the range, we will use one more range function in this for loop, where we specify a minimum of 5 and a maximum of 10.

[Video description begins] *He enters the lines of code, code starts: for num in range (5, 10): print("number:", num). Code ends.* [Video description ends]

And on running this, we see that the range contains all of the integers, 5 through 9.

[Video description begins] *The output reads: number: 5 number: 6 number: 7 number: 8 number: 9.* [Video description ends]

Let us do something a little more with the range function. So this time, we initialize a range between 5 and 8.

[Video description begins] *He enters the lines of code, code starts: for i in range(5, 8): print(i, "square is", i ** 2) print("End of loop"). Code ends.* [Video description ends]

So this should contain the integers 5, 6, and 7. And then within the body of the for loop, we make sure that at each iteration, we print out the value of i along with its square. And then at the end, we just print out that the loop has come to an end. So once we run this, we get the values as well as the squares of 5, 6 and 7.

[Video description begins] *The output reads: 5 square is 25 6 square is 36 7 square is 49 End of loop.* [Video description ends]

One thing we have seen so far with the range function is that when we specify an upper bound of n, then the maximum value in the range which is returned is n minus 1. So if you would like the value n to be part of the range, then we need to make sure that the upper bound is one more than n.

[Video description begins] *He enters the lines of code, code starts: for val in range (5, 10 + 1): print (val) print ("the loop has ended"). Code ends.* [Video description ends]

So here, to ensure that the value 10 is included within the range from 5 to 10, then the upper bound needs to be 11, or 10 + 1, and running this does confirm the fact. So what we have seen so far is a range where the value of each of the elements increases.

[Video description begins] *The output reads: 5 6 7 8 9 10 the loop has ended.* [Video description ends]

But what if we wanted the reverse, where we wanted to start off with a high value and at each iteration, work with a lower value? So here, we will initialize another range, and this is going to be assigned to the variable x, and this should contain the range of integers from 0 through 4.

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

However, when using it in a for loop, we don't use the range directly, but in fact we apply a reverse function on that range. And we'll now print out the values when we perform an iteration over the reversed form of x. And the output gives us the values, 4, 3, 2, 1, and 0. So the reversed function is able to reverse the contents of a sequence, such as a list or a range. Moving along now, let us see if the reversed function can be applied to a string as well, a string being nothing more than a sequence of characters.

[Video description begins] *He enters the lines of code, code starts: for i in reversed("welcome"): print (i). Code ends*. [Video description ends]

So we now iterate each of the characters which make up the string welcome, but in reversed order, and this reversed function works with strings as well.

[Video description begins] *The output reads: e m o c l e w*. [Video description ends]

So one more feature of the range function which we have seen so far is that the sequence of integers which it gives us contains all of the integers within that range.

[Video description begins] *He enters the line of code: list(range(5, 10, 1))*. [Video description ends]

So for example, if we specify a range between 5 and 10, then it gave us the values 5, 6, 7, 8, and 9. In this case, we will call the range function once more, but we specify three arguments.

[Video description begins] *He points to the arguments 5, 10, and 1 in the line of code: list(range(5, 10, 1))*. [Video description ends]

The first is the minimum, the second is the upper bound of the range, and the third is the gap or the interval between each of the integers which are returned. So by default, the value of this interval is 1, which we can confirm by running this cell. And it does give us each of the integers from 5 through 9.

[Video description begins] *The output reads: [5, 6, 7, 8, 9]*. [Video description ends]

But now, we will run this once more, but this time we specify an interval of 2.

[Video description begins] *He enters the lines of code, code starts: x = list(range(5, 10, 2)) print(x). Code ends*. [Video description ends]

And when we do that, what we see that it is every alternate number or every second number in the list which has been returned.

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

So the range function is flexible enough where it returns not only a range of integers, but can also return the integers based on a specific interval or gap. Now this range, once again, need not be an increasing order but can also be in decreasing order. So here, we start off with a value of 5 and then go all the way down to -10.

[Video description begins] *He enters the line of code: tuple(range(5, -10, -2))*. [Video description ends]

But rather than getting every single integer within that range, we will only go down in intervals of -2. Also note that I'm applying the tuple function this time in order to return the sequence as a tuple. But that is not quite relevant for this purpose, you can either convert it to a tuple or a list, or whatever you're comfortable with. What we are interested in is the value of the elements within that sequence.

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

So you see here that the range starts off with the value 5, the second element is two less than the first, and the third element is two less than the second, and so on. And this goes on until we get to the value -9. If you are

familiar with slices in Python, you will notice that this is similar to the start, stop, and step notation of Python slices.

[Video description begins] *He points to the arguments 5, -10, and -2 in the line of code: tuple(range(5, -10, -2))*. [Video description ends]

We will now create another decreasing range which begins at -2 with a lower bound of -10 and in decrements of -2.

[Video description begins] *He enters the line of code: list(range(-2, -10, -2))*. [Video description ends]

So we should expect that the output contains -2, -4, -6 and -8, but will it contain -10? And the answer is no. So for an increasing range, when we specify an upper bound, that is not included within the range itself.

[Video description begins] *The output reads: [-2, -4, -6, -8]*. [Video description ends]

And similarly, for a decreasing range, when we set a lower bound, that won't be included either. Now I had mentioned that this notation for Python ranges is similar to the start, stop, and step notation for Python slices.

[Video description begins] *He enters the lines of code, code starts: start = -2 stop = -10 step = -2 for number in range (start, stop, step): print (number). Code ends*. [Video description ends]

And we can explicitly display that by setting variables start, stop, and step to -2, -10, and -2, and then use them in this range function, which we will use in a for loop. And the output confirms that this does indeed work as a Python slice.

[Video description begins] *The output reads: -2, -4, -6, -8*. [Video description ends]

Exploring the range Function

[Video description begins] *Topic title: Exploring the range Function. The presenter is Kishan Iyer.* [Video description ends]

At the end of the last video, we saw how we can define a decreasing range of values using the range function.

[Video description begins] *He opens the RangeFunctionInForLoop page in the jupyter notebook*. [Video description ends]

We will move back now to an increasing range, where we will iterate over all of the integers in the range between 1 and 10.

[Video description begins] *He enters the lines of code, code starts: for i in range (1, 10): if i % 2 == 0: print("The number %s is divisible by 2" % i) else: print("The number %s is not divisible by 2" % i). Code ends*. [Video description ends]

And then within our for loop, we will make use of an if else block. We check if the number is divisible by 2. And if it is, we print out that fact and even if it's not divisible by 2, we will print the message that the number is not divisible. Now this is similar to a for loop which we had implemented in the previous Jupyter notebook, but on that occasion, we had explicitly defined each of the values in our range of elements. Here we're simplifying matters by making use of the range function. So when we run this we get to iterate over all of the values from 1 through 9.

[Video description begins] *The output displays whether the numbers from 1 to 9 are divisible by 2 or not*. [Video description ends]

Moving along now, we will once again iterate over the values between 1 and 21. And then this time we will check whether the number happens to be divisible by either 3 or by 5.

[Video description begins] *He enters the lines of code, code starts: num_list = [] for i in range(1, 21): if (i%3==0 or i%5==0): num_list.append(i) print(num_list). Code ends.* [Video description ends]

If it is, we will append that value into this list called num_list. And then finally we will print out the contents of that list. And on running this cell, all of the values which are divisible by either 3 or 5 are printed out.

[Video description begins] *The output reads: [3, 5, 6, 9, 10, 12, 15, 18, 20].* [Video description ends]

So now, let us see the true power of the range function. So far the ranges of values which we have been working with have been relatively small. But let's just say a problem involved going over a number of values in a very large range.

[Video description begins] *He enters the lines of code, code starts: items = [] for i in range(100, 301): num_str = str(i) first_digit = int(num_str[0]) second_digit = int(num_str[1]) third_digit = int(num_str[2]) if (first_digit % 2 == 0) and (second_digit % 2 == 0) and (third_digit % 2 == 0): items.append(num_str) print(", ".join(items)). Code ends.* [Video description ends]

So for example, if you needed to identify all the numbers between 100 and 300, which are purely made up of even numbered digits. Then, it's much easier for us to make use of a range function, rather than explicitly define a list containing the 201 integers which we need to iterate over. So here we initialise a for loop and this for loop will iterate over each number in the range 100 to 301. So the last number in this range will be 300. So remember that our task here is to identify all of those numbers which are only made up of even numbered digits. And if we find such a number during our iterations we will load it into an array called items.

So within each iteration we will first convert the number into a string for which we apply the str function on the variable i. And given that we are only working with three digit numbers here, we created three different variables called first_digit, second_digit, and third_digit. The value of first_digit will be the first character in num_string, and we need to convert this to an integer in order to check that it is an even number. We will need to apply a similar transformation on the second digit. So the string form of the second digit is available in num_str at index 1. We convert it to an int and assign it to second_digit.

And similarly, we extract the third digits as well from index 2 of num_str. Now, it is time for us to check that each of these three digits are even numbers. So we need to make three different comparisons within our if statement. So we first check whether first_digit is divisible by 2 for which we use the modulus function, and then we check whether second_digit and third_digits are also divisible by 2. And we make sure that we use the and keyword to check that each and every one of those conditions are satisfied. If that is the case, it means that our number contains only even digits and we will append this number into our items list.

So this will keep going on, until at the end of the for loop the items list will contain all of the numbers which fulfill our criteria. And then we will print out these numbers using this print statement at the bottom. And what we would like in the output is for each of those numbers to be separated by a comma followed by a space. To do that, we use a string literal which contains a comma followed by a space. And then to that we apply the join function. The argument to the join function is our items list. This will print out each of the elements which make up the items list separated by a comma and a space.

[Video description begins] *The output reads: 200, 202, 204, 206, 208, 220, 222, 224, 226, 228, 240, 242, 244, 246, 248, 260, 262, 264, 266, 268, 280, 282, 284, 286, 288.* [Video description ends]

So we run this cell and all of the numbers within our range of 201 numbers which are all made up of even digits have been printed out. So we have now seen how the range function can be used in order to iterate over a range of values. And this applies when we know the upper and lower bounds of the range and also the interval when

required. Let us now take a look at a case of using nested for loops where each of the for loops makes use of the range function.

[Video description begins] *He enters the lines of code, code starts: for num1 in range(1, 10): print("num =", num1, ":", end = " ") for num_2 in range(1, 10): print("{:2d}".format(num1 * num_2), end = " ") print(). Code ends*. [Video description ends]

So let's just say we want to print the multiplication tables for each of the numbers from 1 through 9. For this, you already know that we will need to iterate over all the numbers from 1 through 9. And for each of those numbers, when we produce the multiplication tables, we will be showing the multiples from 1 through 9. This means that we will need to implement nested for loops to accomplish this. In the outer for loop, we iterate over the range of values from 1 through 9. So for that we specify an upper bound of 10.

And then we will print out the value of this number in each iteration of this outer for loop. And then within the inner for loop, we once again iterate over the range between 1 and 10. And we will generate the product of num1 and num2, and then print it out. As an example, consider that in the fifth iteration, the value of num1 will be 5 and we will be producing the multiplication tables for 5. So within the inner for loop, we will first print out 5 times 1 which is 5. In the next iteration we will print out 5 times 2 which is 10, and so on.

At the end of this for loop we make a call to the print function without supplying any arguments, and we do that just to introduce a new line after every iteration of the outer for loop. So let us go ahead and run this cell, and the multiplication tables for all of the integers between 1 and 9 have been printed out. So by now, it should be clear to you how the range function can really simplify the task of performing a specific number of iterations over a defined range of values.

Exercise: Basics of Python for Loops

[Video description begins] *Topic title: Exercise: Basics of Python for Loops. The presenter is Kishan Iyer.* [Video description ends]

In this exercise, you will be writing a fair amount of code. To start off with you will print cubes of all the elements in a given list using a for loop. For loops are very useful control structures in order to perform a repetitive set of actions on a sequence of values. In this case, the sequence of values is a list of integers and the repetitive action is to calculate the cubes of each of those integers. For the second task, you will need to transform this two dimensional list of strings into a flattened or one dimensional list.

This two dimensional list in our example contains two elements, each of which is a one dimensional list. Each of these one dimensional lists represent automobile companies from specific countries. In our case though, we don't care where the companies come from, we just want a single list containing all of the different automobile companies. And you will make use of for loops, in order to create such a list out of this 2-D list.

[Video description begins] *The following lists of the automobile company names are displayed: [['Volkswagen', 'Mercedes', 'BMW'], ['Honda', 'Toyota', 'Mazda']].* [Video description ends]

And finally, you will make use of the range function in order to print all of the leap years which occur in the 21st century. So you will extract all of the leap years from the year 2000 until 2099. We saw similar problems using for loops and range functions throughout this course. So please pause this video, and then spend some time to perform this exercise on your own. For the first task in the exercise, you needed to print out the cubes of the elements in a list of integers.

[Video description begins] *Solution. Cubes of Elements in a List. The following lines of code are displayed, code starts: original_list = [1, 2, 3, 4, 5, 6] cubes_list = [] for i in original_list: cubes_list.append(i ** 3) print(cubes_list). Code ends*. [Video description ends]

So to start off with, consider that we have a list of integers, so here we use an example of the integers from 1 through 6. And then we will also initialize an empty list, in which we will append the values of the cubes of each of the elements in original_list. So in order to calculate cubes for each of these integers, we will need to make use of a for loop, in order to iterate over the elements one by one. So for each integer i in our original_list, we calculate i cubed, and then append it to our cubes_list. Once the for loop terminates, the cubes_list will contain all of the six cube values, and then we can print out those values in order to confirm that we have produced the correct result. So if you were to paste this code in a Jupyter notebook or any other Python IDE, you should get an output exactly like this.

[Video description begins] *The output reads: [1, 8, 27, 64, 125, 216].* [Video description ends]

So the cubes of each of the six integers will be available in our cubes_list. Moving along now to the next task in the exercise, which was to effectively flatten a two dimensional list.

[Video description begins] *Flattening a 2-D List*. [Video description ends]

So given an original_list which contains two one dimensional lists, we will iterate over each of the individual elements, and then add them to this flattened_list.

[Video description begins] *The following lines of code are displayed, code starts: original_list = [['Volkswagen', 'Mercedes', 'BMW'], ['Honda', 'Toyota', 'Mazda']] flattened_list = [] for cars_list in original_list: for car in cars_list: flattened_list.append(car) print(flattened_list). Code ends*. [Video description ends]

So in the end our flattened_list will contain six strings, containing the automobile company names. So here we will setup a for loop, where we will iterate over each of the lists within our original_list. And then in order to iterate over those one dimensional lists, we include a nested for loop within this for loop. And this is where we access each of the individual strings and then append them to our flattened_list. Once all of the four loops have terminated, we will print out the contents of the flattened_list, and you should get an output which looks something like this.

[Video description begins] *The output reads: ['Volkswagen', 'Mercedes', 'BMW', 'Honda', 'Toyota', 'Mazda']*. [Video description ends]

We move along now to the last task in the exercise, where you needed to use the range function in order to print out all of the leap years in the 21st century.

[Video description begins] *The Range Function*. [Video description ends]

So we assume here that the 21st century includes the year 2000, which means the range of years which we need to consider starts at 2000, and ends with 2099. And we also know that it is every fourth year which is a leap year. So how do we create a sequence of all of the leap years in this range? Well, for that, we use the range function. The arguments to this include the lower bound which is included in the range, so that is 2,000. The second argument is the upper bound which is not included in the range and this is 2,100.

And then the third argument is the interval between each of the elements in our range. Given that 2000 is a leap year, and the interval between leap years is every 4 years, the value of our interval will be 4. So this will return a range object which includes the sequence containing all of the leap years. However, in order to convert it to a list, we will need to apply the list function on that range object. And then finally, you can print out the contents of this list in order to view all of the leap years in the 21st century.