Reading: For loops

You've learned about for loops in Python and have explored some examples. For loops are like while loops, but instead of looping continuously until a condition is met, for loops iterate over each element of an iterable sequence, allowing you to perform an action or evaluation with each iteration. This is an important process in computer programming, not just in Python, but in most other languages too. Data professionals use for loops to process data, so it's important for you to familiarize yourself with them as you grow your skills. This reading is a review of the fundamental concepts of for loops.

For loop syntax

A for loop is a control structure that allows you to execute a block of code the same number of times as there are elements in an iterable sequence. You'll learn more about iterable sequences later in this course, but some examples of iterable data types include:

Strings: **'chimichurri'**

Lists: **[1, 2, 3, 4, 5, 6]**

Tuples: **(1, 2, 3, 4, 5)**

Dictionaries: **{'Name': 'Anita', 'Age': 77}**

Sets: **{1, 4, 14, 33}**

Note: The following code block is not interactive.

The basic syntax of a for loop is as follows:

```
for item in iterable_sequence:
  # Code block to be executed for each value in iterable_seque
```

The **iterable_sequence** variable can be any iterable data type, and **item** is a variable whose name is arbitrary —you decide it. However, there are some conventions that you'll encounter when naming this variable. For example, if you're iterating over characters in a string, you'll frequently encounter the variable **char**. If you're iterating over a list of numbers, you'll find **n** or **num**. It's helpful to give this variable a name so readers of your code understand what kind of information is being looped over. So, for a variable called **names** that contains a list of people's names, you might write: **for name in names:**.

A note about the behavior of this variable — its value is reassigned for each iteration of the loop, and it persists even after the loop terminates.

Here's an example:

```
num = 5
y = [1, 2, 3]
for num in y:
  print(num)

print(num)
```

Output:

```
1
2
3
3
```

Notice that **num** exists as a variable before the for loop begins. The for loop's first iteration reassigns its value with that of the first element in the sequence. This reassignment occurs with each iteration of the loop. When the loop terminates, the variable persists, and it contains the value it had after the final iteration of the loop.

The range() function

The **for** loop allows you to create a loop that performs exactly the number of iterations needed for the data structure you're looping over. In other words, whether your iterable sequence contains two, 1,000, or a million elements, you can use the same syntax and don't have to specify the number of iterations you want. However, sometimes you need to perform a task a set number of times, but you don't already have an iterable object to loop over. Or, sometimes you need to generate a known, regular sequence of numbers. This is where the **range()** function is useful.

The **range()** function is a function that takes three arguments: start, stop, step. Its output is an object belonging to the range class. If you only include one argument, it will be interpreted as the stop value. The start and step values by default will be zero and one, respectively. If you include two arguments, they will be interpreted as the start and stop values (again, with step being one by default). Note that the stop value is not included in the range that is returned.

Here are some examples:

A.

```
for i in range(3):
  print(i)
```

Output:

```
0
1
2
```

B.

```
for n in range(2, 5):
  print(n)
```

Output:

```
2
3
4
```

C.

```
for even_num in range(2, 11, 2):
  print(even_num
```

Output:

```
2
4
6
8
10
```

You'll find that the **range()** function is very useful, for example, when creating numbered lists or performing operations on certain indices of an object. You'll learn more about indexing later.

Nested loops

Sometimes you'll need to extract information from nested structures—for example, from a list of lists. One way of doing this is by using nested loops. A nested loop is a loop inside of another loop. You can have an infinite number of nested loops, but it becomes more confusing to read and understand the more nested loops you add.

Here's an example of one loop nested in another:

```
students = [['Igor', 'Sokolov'], ['Riko', 'Miyazaki'], ['Tuva', 'Johansen']]
for student in students:
  for name in student:
     print(name)
  print()
```

Output:

```
Igor
Sokolov

Riko
Miyazaki

Tuva
Johansen
```

In this example, the students variable contains a list of three lists. Each inner list contains two elements: a given name and a surname. The first for loop iterates over the inner lists. The second (nested) for loop iterates over each name in each inner list and prints the name. After each iteration of the outer loop, the program uses an empty print statement to print a new line.

Key takeaways

A **for** loop allows you to execute a block of code the same number of times as there are elements in an iterable sequence. The **range()** function is useful for creating a defined iterable sequence. And nested loops are loops within loops that give you even greater power and control over how your code may execute. These are powerful tools that can be used in many different ways to solve a variety of problems that you'll encounter as a data professional.