Loops Using for

by Sophia

WHAT'S COVERED

In this lesson, you will learn about some of the patterns that generate for loops when coding an algorithm. Specifically, this lesson covers:

1. Revisit the for Loop

We looked at the basics of the `for` loop in a past lesson; however, as a recap, the structure of the `for` loop looks like the following:

EXAMPLE

```
for <variable> in <iterable>:
    <statement(s)>
```

Note: In the example above, the <variable>, <iterable>, and <statement(s)> terms and outside arrows are just for information purposes; these are not keywords or actual code. These are just to explain what goes into each of the parts of a `for` loop.

The <variable> term is initially declared and used to refer to the elements in the iterable.

The <iterable> is the data collection type. Remember the list, tuple, set, and dictionary are all iterable objects.

The <statement(s)> are the statements that will execute for each element in the <iterable>. The <variable> refers to the next element in <iterable> every time that the loop is executed.

Sometimes we want to loop through a collection of things such as a list of words, the lines in a file, or a list of numbers. When we have a collection of things to loop through, we can construct a definite loop using a `for` loop. We call the `while` loop an indefinite loop because it simply loops until its expression's condition becomes False, whereas the `for` loop is looping through a known collection of items or elements. It runs through as many iterations as there are items/elements in the collection.

The syntax of a `for` loop is similar to the `while` loop in that there is a `for` loop statement and then the body of the loop.

EXAMPLE

```
friendsList = ['Joseph', 'Glenn', 'Sally']
for friend in friendsList:
    print('Happy New Year:', friend)
print('Done!')
```

Let's break down what is happening in this `for` loop. First, we define the iterable object, which in this case is a list called `friendsList` that contains three strings (names). The variable `friend` is then assigned to refer to the elements of the list. This variable can be named anything (using best practice variable naming of course) but Python will interpret any variable name placed in that position as the reference to the iterable. The `for` loop goes through the list and executes the body of the loop one time for each of the three strings in the list (each element in the list), resulting in this output:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

Translating this `for` loop to English is not as direct as the `while` loop, but if you think of friends as a collection, it goes like this: "Run the statements in the body of the `for` loop once for each friend in the list named `friendsList`."

Looking at the `for` loop, `for` and `in` are reserved Python keywords, and `friend` and `friendsList` are variables.

```
for friend in friendsList:
    print('Happy New Year:', friend)
```

Here, `friend` is the iteration variable for the `for` loop. The variable `friend` changes for each iteration of the loop and controls when the `for` loop completes. The iteration variable steps successively through the three strings stored in the `friendsList` list.

TRY IT

Directions: Go ahead and enter the program into the IDE and test the `for` loop. Try changing the reference variable `friend` to something else and test it.

```
friendsList = ['Joseph', 'Glenn', 'Sally']
for friend in friendsList:
    print('Happy New Year:', friend)
print('Done!')
```

2. Uses of the for Loop

There are many uses for the `for` loop. These include counting elements within a list, computing totals based on a list, or finding the largest and smallest elements within the list. Any time that we need to look at each element within a list can be a scenario to use a `for` loop.

2a. Counting Elements

To count the number of elements in a list, we would write the following `for` loop.

EXAMPLE

```
countItems = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    countItems = countItems + 1
print('Count: ', countItems)
```

Let's break this loop down. We set the variable `countItems` to zero before the loop starts, then we write a `for` loop to run through the list of numbers.

Did you notice that we did not initialize and create the list before entering the `for` loop?

We initialized the list inside the `for` loop instead, the first line of the `for` loop statement. This is ok to do as we will not be using this list outside of the `for` loop.

KEY CONCEPT

We will discuss scope in a later lesson, but with scope there are two choices, global and local scope. If we wanted to use the list outside of the `for` loop, it's better to create and name the list outside the loop like we have been doing in many past examples; that makes the list at a global scope level and allows the list to be used anywhere in the program. If a list is created inside a function or condition, it will be at a local scope level and can only be used within that function or condition.

For now, let's just use this list inside our loop example. Our iteration variable is named `itervar` (think "iteration variable") and while we do not use `itervar` in the body of the loop directly, it does control the loop and causes the body of the loop to be executed once for each of the elements in the list.

In the body of the loop, we add 1 to the current value of `countItems` for each of the elements in the list (during each iteration of the loop). While the `for` loop is executing, the value of `countItems` is the number of elements we have seen "so far." This means it is keeping count as the elements are being iterated through one

at a time. Remember with iterables, `itervar` (iteration variable) points to the current element and moves through the elements one at a time.

Once the `for` loop completes with the `itervar` variable reaching the last element in the list, the value of `countItems` is the total number of elements. The total number is output at the end of the loop. We constructed this `for` loop so that we have what we want when the loop finishes, which is the count of the number of elements in the list.

TRY IT

Directions: Enter and run the program example and see if you get the same output as below.

```
Count:  6
```

Did you see a count of 6? Try adding more element values in the list and test the output.

2b. Computing Totals

Another similar loop that computes the total of a collection of numbers is as follows.

EXAMPLE

```
totalItems = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    totalItems = totalItems + itervar
print('Total: ', totalItems)
```

Now let's break this one down. We set the variable `totalItems` to zero before the loop starts. In this loop, we do use the iteration variable in the body of the loop. Instead of simply adding one to the `countItems` variable as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running `totalItems` variable during each loop iteration. If you think about the variable `totalItems`, it contains the "running total of the values so far." So before the loop starts, `totalItems` is zero because we have not yet seen any element's values. During a loop, `totalItems` is accumulating the running total. When the loop is finished (all elements have been looped), `totalItems` is the overall total of all the element's values in the list.

As the loop executes, `totalItems` accumulates the sum of the elements; a variable used this way is sometimes called an accumulator.

TRY IT

Directions: Enter and run the program example and see if you get the same total as below.

```
Total:  154
```

154 is the total of all the list element's values.

HINT

Neither of these loop examples, the counter or the total (accumulator), are particularly useful in practice because there are built-in functions `len()` and `sum()` that compute the number of elements in a list and the total of the elements in the list, respectively. If you recall, we discussed these functions in an earlier lesson when we talked about general built-in functions and methods provided by Python.

2c. Finding the Largest Value

To find the largest value in a list or other data collection type, we can construct the following loop.

EXAMPLE

```
largestValue = None
print('Before:', largestValue)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largestValue is None or itervar > largestValue :
        largestValue = itervar
    print('Loop:', itervar, largestValue)
print('Largest:', largestValue)
```

When the program executes, the output is as follows.

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

Let's break down this program. In the program, the variable `largestValue` is best thought of as the "largest value we have seen so far." Before the loop, we set `largestValue` to the constant None. None is a special constant value that we can store in a variable to mark the variable as "empty."

Before the loop starts, the largest value we have seen so far is None, since we have not yet seen any element's values. While the `for` loop is executing, if `largestValue` is None, then we take the first element's value in the first iteration as the largest so far. We can see, in the first iteration, that the value of `itervar` is 3; since `largestValue` is None, we immediately set `largestValue` to be 3.

After the first iteration, largest is no longer None, so the second part of the compound conditional statement (or) checks to see if `itervar` is greater than `largestValue`. It will trigger only when we see a value that is larger than the "largest so far." When we see a new "even larger" value, the next statement of the if conditional is executed and `largestValue` will take that new value of `itervar`. You can see in the program output that `largestValue` progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the element's values and the variable `largestValue` now does contain the largest value in the list.

TRY IT

Directions: Try this for yourself. Enter the program into the IDE and run it. Try changing the list values to see how the loop responds to your changes.

2d. Finding the Smallest Value

To compute the smallest number, the code is very similar with one small change.

EXAMPLE

```
smallestValue = None
print('Before:', smallestValue)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallestValue is None or itervar < smallestValue:
        smallestValue = itervar
    print('Loop:', itervar, smallestValue)
print('Smallest:', smallestValue)
```

This program's output looks like this.

```
Before: None
Loop: 3 3
Loop: 41 3
Loop: 12 3
Loop: 9 3
Loop: 74 3
Loop: 15 3
Smallest: 3
```

Again, the variable `smallestValue` is the "smallest so far" before, during, and after the loop executes. When the loop has completed, `smallestValue` contains the minimum value in the list.

TRY IT

Directions: Try this smallest value program. Try changing the list values, and moving the largest value around. Does it still find the smallest value at the end?

Note as well that similar to the `while` loop, we can also make use of the `break`, `continue` and else statements and they will work in the same way. The `break` statement will immediately end the `for` loop. The `continue` statement will end the current iteration of the loop and go back to the start again to evaluate the expression's condition. The else statement is only executed as long as the loop finishes normally without being ended a `break` statement.

Here's an example of one program that can be used with a `break` statement. If a number is found, we want to output that the number is found and exit the loop, as it doesn't make sense to continue to loop through each element. If a number is not found within a list of numbers, we should output "Number not found" using the else statement. We'll use 12 as an example to set to the variable `numberToFind`. As we iterate through each element in the list, we compare the `itervar` with the value in `numberToFind`. If the number is found, "Number found!" is output to the screen, and the `break` statement is called. If the number is not within the list, the loop exits normally and the "Number not found" is output to the screen.

EXAMPLE

```
numberToFind = 12
for itervar in [3, 41, 12, 9, 74, 15]:
   if itervar == numberToFind:
       print('Number found!')
       break
else:
 print('Number not found.')
```

The output with the number found.

```
Number found!
```

As expected, 12 was found in the list (third element). Once found, the `break` statement stopped the loop and skipped the else statement.

Now, here is the same example but with the value of `numberToFind` not within the list.

EXAMPLE

```
numberToFind = 50
for itervar in [3, 41, 12, 9, 74, 15]:
   if itervar == numberToFind:
       print('Number found!')
       break
else:
 print('Number not found.')
```

The output with the number not found.

```
Number not found.
```

That is correct; 50 is not an element's value in the list and as such, we see the "Number not found" output from the else statement.

SUMMARY

In this lesson, we **revisited the for loop** in more detail including covering its properties as a definite loop. We **used the for loop** to both **count elements** and **compute total** elements of a list. We also saw how a `for` loop can be used to **find the largest or smallest value** in a list. Finally, we learned that a `for` loop can also use the `continue` and `break` statements to change the flow of the loop.

Best of luck in your learning!