

Chapter 5 - Loops

Every programming language I have tried has some kind of looping construct. Most have more than one. The Python world has two types of loops:

- the **for** loop and
- the **while** loop

You will find that the **for** loop is by far the most popular of the two. Loops are used when you want to do something many times. Usually you will find that you need to do some operation or a set of operations on a piece of data over and over. This is where loops come in. They make it really easy to apply this sort of logic to your data.

Let's get started learning how these fun structures work!

The for Loop

As mentioned above, you use a loop when you want to iterate over something n number of times. It's a little easier to understand if we see an example. Let's use Python's builtin **range** function. The range function will create a list that is n in length. In Python 2.x, there is actually another function called **xrange** that is a number generator and isn't as resource intensive as range. They basically changed **xrange** into **range** in Python 3. Here is an example:

```
>>> range(5)
range(0, 5)
```

As you can see, the range function above took an integer and returned a **range** object. The range function also accepts a beginning value, an end value and a step value. Here are two more examples:

```
>>> range(5,10)
range(5, 10)
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
```

The first example demonstrates that you can pass a beginning and end value and the range function will return the numbers from the beginning value up to but not including the end value. So in the case of 5-10, we get 5-9. The second example shows how to use the **list**

function to cause the range function to return every second element between 1 and 10. So it starts with one, skips two, etc. Now you're probably wondering what this has to do with loops. Well one easy way to show how a loop works is if we use the range function! Take a look:

```
>>> for number in range(5):  
    print(number)  
  
0  
1  
2  
3  
4
```

What happened here? Let's read it from left to right to figure it out. For each number in a range of 5, print the number. We know that if we call range with a value of 5, it will return a list of 5 elements. So each time through the loop, it prints out each of the elements. The for loop above would be the equivalent of the following:

```
>>> for number in [0, 1, 2, 3, 4]:  
    print(number)
```

The range function just makes it a little bit smaller. The for loop can loop over any kind of Python iterator. We've already seen how it can iterate over a list. Let's see if it can also iterate over a dictionary.

```
>>> a_dict = {"one":1, "two":2, "three":3}  
>>> for key in a_dict:  
    print(key)  
  
three  
two  
one
```

When you use a **for** loop with a dictionary, you'll see that it automatically loops over the keys. We didn't have to say **for key in a_dict.keys()** (although that would have worked too). Python just did the right thing for us. You may be wondering why the keys printed in a different order than they were defined in the dictionary. As you may recall from chapter 3, dictionaries are unordered, so when we iterate over it, the keys could be in any order.

Now if you know that the keys can be sorted, then you can do that before you iterate over them. Let's change the dictionary slightly to see how that works.

```
>>> a_dict = {1:"one", 2:"two", 3:"three"}  
>>> keys = a_dict.keys()  
>>> keys = sorted(keys)  
>>> for key in keys:  
    print(key)  
  
1  
2  
3
```

Let's take a moment to figure out what this code does. First off, we create a dictionary that has integers for keys instead of strings. Then we extract the keys from the dictionary. Whenever you call the `keys()` method, it will return an unordered list of the keys. If you print them out and find them to be in ascending order, then that's just happenstance. Now we have a view of the dictionary's keys that are stored in a variable called `keys`. We sort it and then we use the `for` loop to loop over it.

Now we're ready to make things a little bit more interesting. We are going to loop over a range, but we want to print out only the even numbers. To do this, we want to use a conditional statement instead of using the range's step parameter. Here's one way you could do this:

```
>>> for number in range(10):  
    if number % 2 == 0:  
        print(number)  
  
0  
2  
4  
6  
8
```

You're probably wondering what's going on here. What's up with the percent sign? In Python, the `%` is called a modulus operator. When you use the modulus operator, it will return the remainder. There is no remainder when you divide an even number by two, so we print those numbers out. You probably won't use the modulus operator a lot in the wild, but I have found it useful from time to time.

Now we're ready to learn about the `while` loop.

The `while` Loop

The `while` loop is also used to repeat sections of code, but instead of looping `n` number of times, it will only loop until a specific condition is met. Let's look at a very simple example:

```
>>> i = 0  
>>> while i < 10:  
    print(i)  
    i = i + 1
```

The while loop is kind of like a conditional statement. Here's what this code means: while the variable **i** is less than ten, print it out. Then at the end, we increase **i**'s value by one. If you run this code, it should print out 0-9, each on its own line and then stop. If you remove the piece where we increment **i**'s value, then you'll end up with an infinite loop. This is usually a bad thing. Infinite loops are to be avoided and are known as logic errors.

There is another way to break out of a loop. It is by using the **break** builtin. Let's see how that works:

```
>>> while i < 10:
    print(i)
    if i == 5:
        break
    i += 1

0
1
2
3
4
5
```

In this piece of code, we add a conditional to check if the variable **i** ever equals 5. If it does, then we break out of the loop. As you can see from the sample output, as soon as it reaches 5, the code stops even though we told the while loop to keep looping until it reached 10. You will also note that we changed how we increment the value by using **+=**. This is a handy shortcut that you can also use with other math operations, like subtraction (**-=**) and multiplication (***=**).

The **break** builtin is known as a **flow control tool**. There is another one called **continue** that is used to basically skip an iteration or continue with the next iteration. Here's one way to use it:

```
i = 0

while i < 10:
    if i == 3:
        i += 1
        continue

    print(i)

    if i == 5:
        break
    i += 1
```

This is a little confusing, no? Basically we added a second conditional that checks if **i** equals 3. If it does, we increment the variable and continue with the next loop, which effectively skips printing the value 3 to the screen. As before, when we reach a value of 5, we break out of the

loop.

There's one more topic we need to cover regarding loops and that's the **else** statement.

What else is for in loops

The **else** statement in loops only executes if the loop completes successfully. The primary use of the **else** statement is for searching for items:

```
my_list = [1, 2, 3, 4, 5]

for i in my_list:
    if i == 3:
        print("Item found!")
        break
    print(i)
else:
    print("Item not found!")
```

In this code, we break out of the loop when **i** equals 3. This causes the **else** statement to be skipped. If you want to experiment, you can change the conditional to look for a value that's not in the list, which will cause the **else** statement to execute. To be honest, I have never seen anyone use this structure in all my years as a programmer. Most of the examples I have seen are bloggers trying to explain what it is used for. I have seen several who use it to raise an error if an item is not found in the iterable that you were searching. You can read a fairly in depth article by one of the Python core developers [here](#).

Wrapping Up

Hopefully at this point you can see the value in Python loops. They make repetition easier and pretty easy to understand. You will likely see the **for** loop much more often than the **while** loop. In fact, we are going to look at another way **for** loops are used in the next chapter when we learn about comprehensions! If you're still not quite sure how all this works, you may want to re-read this chapter before continuing.