Python Essentials (Pt. 3)

Overview: This time, we will be covering for loops, while loops, the range function, and the basics of function!

For Loops
For loops are important in iterating over several values in an object. Overall, they make it more convenient and minimizes clutter in the code.

There are several different "formats" that a for loop can take. The most common is perhaps the "for i in range()" format. This allows you to repeat a block of code for a set amount of time; of course, all for loops do this, but this is the most generic form--others are specific to the type of object you are using. In the following example, the for loops will go over each individual value in the sequence.

It is interesting to note, however, that range() is a built-in Python function and not a part of the for loop format itself; if you have a list or a string you wanted to iterate through, for example, you would use a different for loop format that is more efficient. The Range() function is explained in greater detail below.

```
Input: num = 6
       for i in range(num): // will loop through 6 times
           print(i)
Output: 0
        1
        2
        3
        4
        5
```

(Note: remember that computer science always starts counting from zero instead of one unless specified to do otherwise.)

This next one is the more efficient, specifically for a sequence for loop (i.e. lists, tuples, dictionaries), although the for loop for iterating through Strings looks pretty much the same. The difference between this one and the one above is that "for i in range()" only takes the index--therefore, if you wanted to access the specific character or thing at that index you would have to retrieve it--but the sequence for loop directly accesses the value at each consecutive index.

Try plugging this code into your Command Prompt/Anaconda and see for yourself what happens!

```
Input: seq = [1, 2, 3, 4, 5]
       for num in seq:
             print(num) // Remember to keep the correct indexing and
```

```
                              white space in your code! This allows for
                              organization and makes it easier to read.
Output: 1
        2
        3
        4
        5 // notice how it will print on a new line each time
```

Another example of the use of the for loop can be manipulating lists.

```
Input: x = [1, 2, 3]
       y = [ ] // At the moment, the list y is empty.
       for num in x:
            y.append(num**3) // The append function was covered in
the
                                previous lesson, but it simply adds
                                values to the end of the list.
                                Additionally, ** indicates 'to the
                                exponent of.'
       print(y)
Output: [1, 8, 27]
```

You can also have nested for loops--which basically means a for loop inside of a for loop. Generally, you'd do this if you are doing something like a sorting algorithm or working with 2-D arrays (more advanced concepts that will hopefully be covered soon!). A more simple use of nested for loops would be if you were iterating through a list of lists. Here:

```
Input: list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
       for list in list_of_lists: // accesses each list
           for x in list:         // accesses each value in the list
                print(x)
Output: 1
        2
        3
        4
        5
        6
        7
        8
        9
```

While Loops

While loops are generally used when you don't know how long you want to repeat a block of code, since a while loop will run as long as a certain condition is met. If we don't code the proper steps to end the loop, it will continue forever and probably cause your program to crash. So, remember to add a condition or include a break statement!

```
Input: i = 0 // i stands for index in this case
       while i < 4:
            print('i is equal to: { }'.format(i))
            // The format() function in Python allows us to substitute
                 values into the placeholder, and the parameter is
                 passed in the parentheses.
            i += 1 // This will ensure that the loop won't continue
                      forever, because it will eventually equal 4,
                      making the condition false.
Output: i is: 0
        i is: 1
        i is: 2
        i is: 3
```

As I mentioned before, you could also include a break statement in your code. This exits the loop at a specific value, even if the condition stated in the while loop is still true. Using the same example as before, we have:

```
Input: i = 0
       while i < 4
            print(i) // The only difference between this print
                      statement and the one above is that this no
                      longer has the preceding words "i is equal to:
                      ".
            if i == 2:
                 break // As you can see, even though 2 < 4, we can
                      break the loop using this simple syntax.
            i += 1
Output: 0
        1
        2
```

(Note: because the "if i == 2" occurs after *the print statement, 2 still gets printed before you break out of the while loop.)*

Range

The range() function in Python will return a sequence of integers within the given parameters. It increases automatically by 1, and you can define the beginning and end points. If you remember lists from the previous lesson, this is actually very similar! The first value is included, but the second is excluded.

```
Input:  list(range(0,6))
```

```
Output: [0, 1, 2, 3, 4, 5] // The range automatically starts at 0
                           unless otherwise specified. So, this is
                      the
                           same output as if we had written as:
                           list(range(6)).
```

Here is another example of the range function in conjunction with for loops, but with a twist!

```
Input: for value in range(1,3):
           print(value)
Output: 1
        2
```

You don't always have to start from 0; the format above starts from zero and goes up to three, exclusive. Think about for i in range() as "for value in range(beginning value-i, end value-e)"--the i standing for inclusive and e for exclusive.

Functions

All the stuff we have written so far would either be in the main() function or typed directly into the program (not in any function), as Python, unlike many other languages, does not really have a main method. To create a main method, you would do something like this:

```
def main():
      print("bleh")
```

However, to make sure that the main() method executes, you have to include these few lines of code:

```
if __name__== "__main__":
    main()
```

"__name__" is a special variable in Python that basically tells you how the code is being executed--whether it's being imported or run directly. When code is run directly, __name__ is automatically set to __main__; when this occurs, __name__ == "__main__" will be true and then execute the main method through "main()", which calls the main function. Technically, you could just call main() after you write the main method, like this--

```
def main():
      print("bleh")

main()
```

And it would work the same, but the official way is using the if __name__ statement.

Is this confusing? It should be, but hopefully not for long! The main function is a type of **function**, which is, in very simple terms, a block of code that will only run if called. Functions are useful because it means

there's less code in your main function. Of course, I did say that Python doesn't really have a specific main method, but it makes organizing your code much easier, and it's good practice.

To create a function, you use "def ___()", the blank being the name of your method, similar to how we created the main method. Let's say you are studying for physics and you are reviewing Newton's Laws of Motion, among other things. Instead of having three print statements in the main method, which might get mixed up in all the other print statements displaying different bits of knowledge, you could create a function specifically for Newton's Laws of Motion, like this:

```
def newtonsLawsOfMotion():
      print("an object in motion will stay in motion and an object at
rest will stay at rest unless acted upon by an unbalanced force")
      print("force is equal to the mass of an object times its
acceleration")
      print("every action has an equal and opposite reaction")
```

Then, to call on this function in the main method, you only need to call it by name like you would call a person--

```
def main():
      newtonsLawsOfMotion()
```

The parentheses are important because sometimes, functions will take **parameters**; this is basically information that the function requires to run but comes from an external source. For example, if you had a multiplication function that multiplies two numbers, you would need to give the function said two numbers in its parameter so that it can return the correct answer.

```
def multiply(num1, num2):
      product = num1 * num2
      return product

def main():
      num1 = 4
      num2 = 5
      print(multiply(num1, num2))
```

This concludes the third lesson. Try some of this on your own, especially the functions. They will make more sense the more you try them out. Next lesson will be out sometime next week or the week after.
-Emma and Nicole