Assignment 4: Loops

Loops

Imagine you were asked to write your name down 100 times in a row. This might take you a long time, and you might make a few mistakes along the way. This is a perfect task for a computer, which would be able to do it really fast and without any mistakes. You can take advantage of this by using loops. A loop is a block of code that will repeat over and over again.

There are two types of loops, “while loops” and “for loops”. While loops will repeat while a condition is true, and for loops will repeat a certain number of times.

Section 1: For Loops

A for loop is used when the number of times we want to run a piece of code is predetermined by either the programmer or by an existing variable.

This is what a for loop looks like:

```
for i in range(4):
    print(i)
```

Let's break down what this code snippet does.

for - Every for loop must start with this keyword.

i - simply represents one of the values in the data that we are looping through.

> It doesn't have to be i (although that is commonly used by programmers). Instead we could use any other variable name

in - is another keyword that is always used with for loops. It essentially tells Python what we are looping through

range(4) - is a simple function that expands into [0,1,2,3]. It creates a list starting a 0 and ending at the specified number (non-inclusive). We'll expand upon this in a little bit.

The header of for loops end with a colon (:). This creates a dividing point between the loop header and the loop body.

print(i) - will print the value of i which was declared in the loop header. Recall that the loop variable could be named nearly anything. If the variable was named number, then we would have used print(number). As the loop progresses, the value of i changes to each of the corresponding values in the data structure specified (range(4) in this instance). Please note that the loop body is indented from the loop header, this is required

range() Function:

The range is quite useful with for loops as they allow you to count. This function takes on three forms:

1. range(stop) - Creates a list from 0 to stop (non-inclusive) and counts by one.

For example range(2) would create a list [0,1,2].

2. range(start, stop) - Creates a list from start to stop (non-inclusive) and counts by one.

For example range(2, 8) would create a list [2,3,4,5,6,7,8].

3. range(start, stop, interval)- Creates a list from start to stop (non-inclusive) and counts by a specified interval.

For example range(2, 8, 2) would create a list [2, 4, 6].

Looping Through Lists:

You can also use for loops to loop through lists as well.

There are two different ways we can do this one is the way we have been using loops and another “fancy” way to do it without the range function.

Let’s look at the normal way first

```
items = ["file", "steam", "water", "light"]

for index in range(len(items)):
    print(items[index])
```

What’s going on here?

We use the range function in order to tell the program how many time we want this llop tor run. In our case we want it to run the length of the loop, hence why we use the len() function to get the length of the items list.

Remember index is keeping track of where we are in the loop. It starts at 0 then goes to 1 etc. Finally the print statement print the items located at the index.

Now the fancy way

```
items = ["file", "steam", "water", "light"]

for item in items:
    print(item)
```

What’s going on here?

This is a special way to loop through lists where you don’t have to keep track of the index or use the range function.

The code reads as follows “for every item in the list items do something”. So instead of worry about the index of something this version just gives us the 1st element of the list, then the 2nd and so on.

All you have to do is print(item) because item is the element in the list you don’t need to access it with an index.

It good to know both versions as one will often times be more helpful than the other depending on the task

A Note on the Importance of Indentation

Look at the following two functions

```
def function_1():
    for i in range(10):
        if i == 4:
            print('We made it to 4!')
        return i

print(function_1())

# Output is 0

def function_2():
    for i in range(10):
        if i == 4:
            print('We made it to 4!')
            return i

print(function_2())

# Output is:
"We made it to 4!"
4
```

The output of the first function is 0. But it should be 4 right? Well if we look at how a for loop looks we will find the error. The for loop should run 10 times, but once it recahes the i == 4 run we should return.

Rember if you have a return statement anywhere in a function it will automatically end what ever is happening and return.

So looking at the first function because the return statement is not under indented in the if statement when the loop runs for the 1st time it skips over the if statement becuase it's false. Then it hits the return statement and stops stops the function returning 0.

In function 2 because the return statement is in the if statement the program will only return something when that if condition is satisfied.

The example is meant to show how important indentation is to your code. You need to pay close attention to ensure you get the results that you expect.

The example above is not a typical use case for while. Frankly, it would be better to use a for loop in this case, however this is simply a demonstration.

- Similar to for loops, while loops must also start with the while keyword.

- We next put our end condition for the loop. In order for the body of the loop to run, this condition must be True. If we put while True: the loop would typically never end, which is known as an *infinite loop*. **You need to be careful about potentially creating infinite loops, as they can potentially cause your computer or Colab to crash.** Since we initially set x=1, and 1 < 4 is True, the loop starts.

- print(x) and x += 1 are the loop body. **Notice that the loop body is indented.** We do x += 1 to increment the value of x to ensure that the loop eventually terminates.

break Statement:

Sometimes it is useful to terminate a loop prematurely. The break statement is used to achieve this.

Consider the following example:

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

This loop would print:

0

1

2

This is just a standard for loop that will print the values of i. However, once i == 3, the loop will terminate, and your program will continue executing the code that follows the loop.

continue Statement:

continue is a little different compared to break. Instead of terminating the whole loop, it just ends that iteration.

Consider the following example:

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

This loop would print:

0

1

2

4

Notice how the loop continues after reaching the continue. All the continue did in this instance is skip the print(i) statement when i == 3.

Practice Time!

You have 3 exercises involving loops that you'll need to complete.

They will be in the loops.ipynb file in Brighspace module 4.

Submit the completed exercises on gradescope.