Loops in Python

Loops are important in any programming language as they help you to execute a block of code repeatedly. You will often come face to face with situations where you would need to use a piece of code over and over but you don't want to write the same line of code multiple times.

In Python We have

- **For Loop**
- **While Loop**

for Loop

A `for` loop acts as an iterator in Python; it goes through items that are in a *sequence* or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

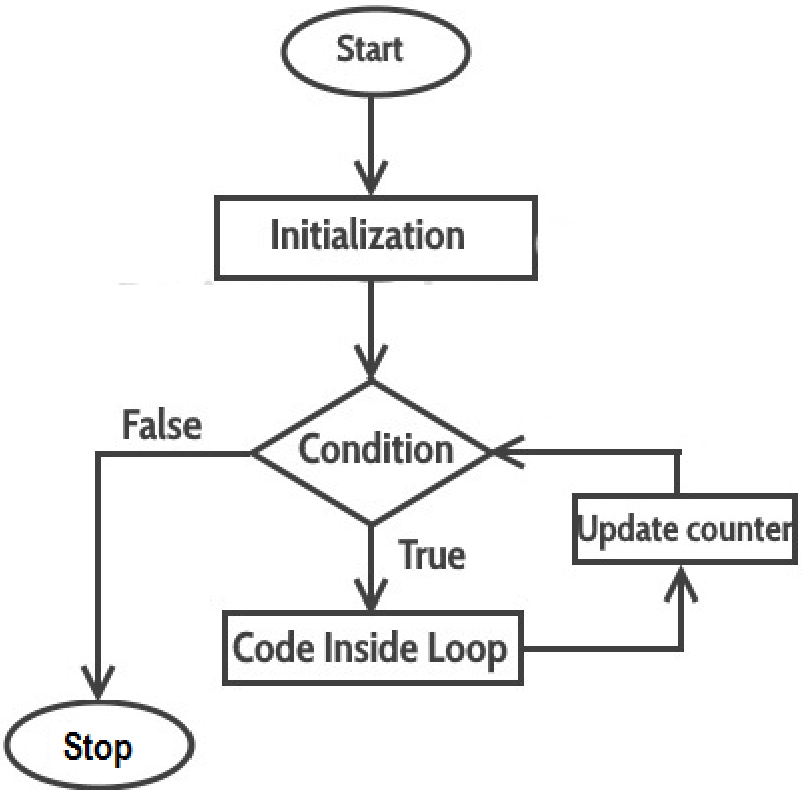
We've already seen the `for` statement a little bit in past lectures but now let's formalize our understanding.

Here's the general format for a `for` loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use `if` statements to perform checks.

Let's go ahead and work through several example of `for` loops using a variety of data object types. We'll start simple and build more complexity later on.



Example 1

Iterating through a list

In [1]:

```python
# We'll learn how to automate this sort of list in the next lecture
list1 = [1,2,3,4,5,6,7,8,9,10]
```

In [2]:

```python
for num in list1:
    print(num)
```

```
1
2
3
4
5
6
7
8
9
10
```

Notice that if a number is fully divisible with no remainder, the result of the modulo call is 0. We can use this to test for even numbers, since if a number modulo 2 is equal to 0, that means it is an even number!

Back to the `for` loops!

Example 2

Let's print only the even numbers from that list!

In [3]:

```python
for num in list1:
    if num % 2 == 0:
        print(num)
```

```
2
4
6
8
10
```

We could have also put an `else` statement in there:

In [4]:

```python
for num in list1:
    if num % 2 == 0:
        print(num)
    else:
        print('Odd number')
```

```
Odd number
2
Odd number
4
Odd number
6
Odd number
8
Odd number
10
```

Example 3

Another common idea during a `for` loop is keeping some sort of running tally during multiple loops. For example, let's create a `for` loop that sums up the list:

In [5]:

```python
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum = list_sum + num

print(list_sum)
```

```
55
```

Great! Read over the above cell and make sure you understand fully what is going on. Also we could have implemented a `+=` to perform the addition towards the sum. For example:

In [6]:

```python
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum += num

print(list_sum)
```

```
55
```

Example 4

We've used `for` loops with lists, how about with strings? Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

In [7]:

```python
for letter in 'This is a string.':
    print(letter)
```

```
T
h
i
s
 
i
s
 
a
 
s
t
r
i
n
g
.
```

Example 5

Let's now look at how a `for` loop can be used with a tuple:

In [8]:

```python
tup = (1,2,3,4,5)

for t in tup:
    print(t)
```

```
1
2
3
4
5
```

Example 6

Tuples have a special quality when it comes to `for` loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of *tuple unpacking*. During the `for` loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

In [9]:

```python
list2 = [(2,4),(6,8),(10,12)]
```

In [10]:

```python
for tup in list2:
    print(tup)
```

```
(2, 4)
(6, 8)
(10, 12)
```

In [11]:

```python
# Now with unpacking!
for (t1,t2) in list2:
    print(t1)
```

```
2
6
10
```

Cool! With tuples in a sequence we can access the items inside of them through unpacking! The reason this is important is because many objects will deliver their iterables through tuples. Let's start exploring iterating through Dictionaries to explore this further!

Example 7

In [12]:

```python
d = {'k1':1,'k2':2,'k3':3}
```

In [13]:

```python
for item in d:
    print(item)
```

```
k1
k2
k3
```

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

We're going to introduce three new Dictionary methods: **.keys()**, **.values()** and **.items()**

In Python each of these methods return a *dictionary view object*. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a view. Let's see it in action:

In [14]:

```python
# Create a dictionary view object
d.items()
```

Out[14]:

```
dict_items([('k1', 1), ('k2', 2), ('k3', 3)])
```

Since the .items() method supports iteration, we can perform *dictionary unpacking* to separate keys and values just as we did in the previous examples.

In [15]:

```python
# Dictionary unpacking
for k,v in d.items():
    print(k)
    print(v)
```

```
k1
1
k2
2
k3
3
```

If you want to obtain a true list of keys, values, or key/value tuples, you can *cast* the view as a list:

In [16]:

```
list(d.keys())
```

Out[16]:

```
['k1', 'k2', 'k3']
```

Remember that dictionaries are unordered, and that keys and values come back in arbitrary order. You can obtain a sorted list using sorted():

In [17]:

```
sorted(d.values())
```

Out[17]:

```
[1, 2, 3]
```

while Loop

The `while` statement in Python is one of most general ways to perform iteration. A `while` statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statements
else:
    final code statements
```

Let's look at a few simple `while` loops in action.

In [18]:

```
x = 0

while x < 10:
    print(x)
    x+=1
```

```
0
1
2
3
4
5
6
7
8
9
```

Notice how many times the print statements occurred and how the `while` loop kept going until the True condition was met, which occurred once x==10. It's important to note that once this occurred the code stopped. Let's see how we could add an `else` statement:

In [19]:

```python
x = 0

while x < 10:
    print(x)
    x+=1

else:
    print('All Done!')
```

```
0
1
2
3
4
5
6
7
8
9
All Done!
```

break, continue, pass

We can use `break` , `continue` , and `pass` statements in our loops to add additional functionality for various cases. The three statements are defined by:

```
break: Breaks out of the current closest enclosing loop.
continue: Goes to the top of the closest enclosing loop.
pass: Does nothing at all.
```

Thinking about `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

`break` and `continue` statements can appear anywhere inside the loop’s body, but we will usually put them further nested in conjunction with an `if` statement to perform an action based on some condition.

Let's go ahead and look at some examples!

In [20]:

```python
x = 0

while x < 10:
    print(x)
    x+=1
    if x==3:
        print('x==3')
    else:
        print('continuing...')
        continue
```

```
0
continuing...
1
continuing...
2
x==3
3
continuing...
4
continuing...
5
continuing...
6
continuing...
7
continuing...
8
continuing...
9
continuing...
```

Note how we have a printed statement when x==3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x ==3 and see if the result makes sense:

In [21]:

```
x = 0

while x < 10:
    print(x)
    x+=1
    if x==3:
        print('Breaking because x==3')
        break
    else:
        print('continuing...')
        continue
```

```
0
continuing...
1
continuing...
2
Breaking because x==3
```

Note how the other `else` statement wasn't reached and continuing was never printed!

After these brief but simple examples, you should feel comfortable using `while` statements in your code.

A word of caution however! It is possible to create an infinitely running loop with `while` statements. For example:

In [ ]:

```
# DO NOT RUN THIS CODE!!!!
while True:
    print("I'm stuck in an infinite loop!")
```

```
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
I'm stuck in an infinite loop!
```

A quick note: If you *did* run the above cell, click on the Kernel menu above to restart the kernel!

Conclusion

We've learned how to use for loops to iterate through tuples, lists, strings, and dictionaries. It will be an important tool for us, so make sure you know it well and understood the above examples.