for Loops

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 the keys or values.

We've already seen the **for** statement a little bit in past lectures but now lets 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]:
```
# We'll learn how to automate this sort of list in the next lecture
l = [1,2,3,4,5,6,7,8,9,10]
```

In [5]:
```
for num in l:
    print num
```

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

Great! Hopefully this makes sense. Now lets add a if statement to check for even numbers. We'll first introduce a new concept here--the modulo.

Modulo

The modulo allows us to get the remainder in a division and uses the % symbol. For example:

In [5]:
```
17 % 5
```

Out[5]:
```
2
```

This makes sense since 17 divided by 5 is 3 remainder 2. Let's see a few more quick examples:

In [6]:
```
# 3 Remainder 1
10 % 3
```

Out[6]:
```
1
```

In [9]:
```
# 2 Remainder 4
18 % 7
```

Out[9]:
```
4
```

In [10]:
```
# 2 no remainder
4 % 2
```

Out[10]:
```
0
```

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 [11]:
```
for num in l:
    if num % 2 == 0:
        print num
```

```
2
4
6
8
10
```

We could have also put in else statement in there:

In [12]:
```
for num in l:
    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 the multiple loops. For example, lets create a for loop that sums up the list:

In [13]:
```
# Start sum at zero
list_sum = 0

for num in l:
    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 to the addition towards the sum. For example:

In [14]:
```
# Start sum at zero
list_sum = 0

for num in l:
    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 [6]:
```
for letter in 'SAURABH':
    print letter
```

```
S
A
U
R
A
B
H
```

##Example 5 Let's now look at how a for loop can be used with a tuple:

In [16]:
```
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 [17]:
```
l = [(2,4),(6,8),(10,12)]
```

In [18]:
```
for tup in l:
    print tup
```

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

In [19]:
```
# Now with unpacking!
for (t1,t2) in l:
    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 object will deliver their iterables through tuples. Let's start exploring iterating through Dictionaries to explore this further!

##Example 7

In [1]:
```
d = {'k1':1,'k2':2,'k3':3}
```

In [2]:
```
for item in d:
    print item
```

```
k3
k2
k1
```

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

Here is where we are going to have a Python 3 Alert!

Python 3 Alert!

Python 2: Use .iteritems() to iterate through

In Python 2 you should use .iteritems() to iterate through the keys and values of a dictionary. This basically creates a generator (we will get into generators later on in the course) that will generate the keys and values of your dictionary. Let's see it in action:

In [9]:
```
# Creates a generator
d.iteritems()
```

Out[9]:
```
<dictionary-itemiterator at 0x104365520>
```

Calling the items() method returns a list of tuples. Now we can iterate through them just as we did in the previous examples.

In [10]:
```
# Create a generator
for k,v in d.iteritems():
    print k
    print v
```

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

Python 3: items()

In Python 3 you should use .items() to iterate through the keys and values of a dictionary. For example:

In [11]:
```
# For Python 3
for k,v in d.items():
    print(k)
    print(v)
```

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

You might be wondering why this worked in Python 2. This is because of the introduction of generators to Python during its earlier years. (We will go over generators and what they are in a future section, but the basic notion is that generators don't store data in memory, but instead just yield it to you as it goes through an iterable item).

Originally, Python items() built a real list of tuples and returned that. That could potentially take a lot of extra memory.

Then, generators were introduced to the language in general, and that method was reimplemented as an iterator-generator method named iteritems(). The original remains for backwards compatibility.

One of Python 3's changes is that items() now return iterators, and a list is never fully built. The iteritems() method is also gone, since items() now works like iteritems() in Python 2.

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.

More resources (http://www.tutorialspoint.com/python/python_for_loop.htm)