حلقه‌های for

یک حلقه `for` برای تکرار کردن یک عملیات در طول یک لیست یا تکرار شونده‌ها استفاده می‌شود. یعنی در هر دور خود یک عضو از یک **دنباله** را استفاده می کند. بگذارید ساختار این حلقه را فرمول بندی کنیم:

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

این که برای متغیر item چه اسمی بگذاریم، به برنامه نویس بستگی دارد، اما باز هم بهتر است که اسمی انتخاب شود که مفهوم داشته باشد. که وقتی دوباره کد را می‌بینید متوجه شوید! این متغیر می‌تواند درون حلقه نیز بکار رود، مثلا گذاشتن یک شرط روی آن. حالا بگذارید یک سری مثال را با هم ببینیم:

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:
    if num % 2 == 0:
        print(num)
```

```
2
4
6
8
10
```

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

In [3]:
```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 [4]:
```
# Start sum at zero
list_sum = 0
for num in list1:
    list_sum = list_sum + num
print(list_sum)
```
```
55
```

In [5]:
```
sum(list1)
```
Out[5]:
```
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]:
```
# 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]:
```
for letter in 'This is a string.':
    print(letter, end=' ')
```
```
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]:
```
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]: list2 = [(2,4),(6,8),(10,12)]
```

```
In [10]: for tup in list2:
             print(tup)
```

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

```
In [15]: # Now with unpacking!
         for (t1, t2) in list2:
             print(t2+t1)
```

```
6
14
22
```

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

```
In [18]: for item in d.values():
             print(item)
```

```
1
2
3
```

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 [19]: # Create a dictionary view object
         d.items()
```

```
Out[19]: 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 [20]:
```
# 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 [21]:
```
list(d.keys())
```
Out[21]:
```
['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 [22]:
```
sorted(d.values())
```
Out[22]:
```
[1, 2, 3]
```

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

In [23]:
```
list3 = []
for num in [1,2,3,4,5,6,7,8,9]:
    if num % 3 == 0:
        list3.append(num)
print(list3)
```
```
[3, 6, 9]
```

In [24]:
```
range(1, 10)
```
Out[24]:
```
range(1, 10)
```

In [26]:
```
list(range(1,10))
```
Out[26]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [28]:
```
for num in range(2, 10,2):
    print(num)
```
```
2
4
6
8
```

In [29]:
```python
list4 = []
for num in range(1,10):
    if num % 3 == 0:
        list4.append(num)
```

In [30]:
```python
print(list4)
```

```
[3, 6, 9]
```

MATLAB Code:

```
for num = start:step:stop
    statements to do
end
```

Python code:

```
for num in sequence:
   statement to do
```

Python code:

```
for num in range(start, stop+1, step):
   statement to do
```

In [ ]:
```python

```