Data Structures

Python, unlike C, C++, and Java, has a number of *built-in* data structures. We will go through several of them here.

Lists

Lists are one of the most basic data structures, and are often used as building blocks for more complex data structures. Lists in Python are *sequence* types: they have a specified order that is preserved when you, for example, use a `for` loop to access the list.

You can create lists using very simple notation:

```
In [1]: list1 = [0, 2, 4, 6, 8]
```

```
In [2]: print(list1)
```

```
[0, 2, 4, 6, 8]
```

And, of course, Python does not care what type of elements you put in the list:

```
In [3]: list1 = ["a", "b", "c"]
        print(list1)
```

```
['a', 'b', 'c']
```

```
In [4]: list1 = ["a", 1, 2]
        print(list1)
```

```
['a', 1, 2]
```

You can also instantiate an empty list:

```
In [5]: list1 = []
        print(list1)
```

```
[]
```

Or use some syntactic trickery to instantiate a list of a certain size, with default elements:

```
In [6]: list1 = 5 * [0]
        list2 = 5 * ['a']
        print(list1)
        print(list2)
```

```
[0, 0, 0, 0, 0]
['a', 'a', 'a', 'a', 'a']
```

Although in many ways lists behave like arrays, lists can be changed. You can add elements to the end of the list:

```
In [7]: list1.append(2)
        print(list1)
```

```
[0, 0, 0, 0, 0, 2]
```

Or add elements to the middle of the list:

```
In [8]: list1.insert(2, 5)
        print(list1)
```

```
[0, 0, 5, 0, 0, 0, 2]
```

Remove an element from the list (in this case, the fourth element)

```
In [9]: list1.pop(3)
        print(list1)
```

```
[0, 0, 5, 0, 0, 2]
```

Or even remove the first element from the list that has a particular value:

```
In [10]: list1.remove(5)
         print(list1)
```

```
[0, 0, 0, 0, 2]
```

You can access elements in a list just like they are arrays, or iterate over them

```
In [11]: print(list1[3])
```

```
0
```

In [12]:
```
for i in list1 :
    print(i)
```

```
0
0
0
0
2
```

Note that when a variable is referring to a list in Python (like almost all other variables except basic types like `int` s), that variable contains a *reference* (think "pointer") to the list; it does not talk about the list itself. This can lead to some interesting effects:

In [13]:
```
list2 = list1
print(list2)
list2.append(7)
print(list2)
print(list1) #this also changes!
```

```
[0, 0, 0, 0, 2]
[0, 0, 0, 0, 2, 7]
[0, 0, 0, 0, 2, 7]
```

What has happened is that `list2` and `list1` both refer to the same actual list in memory, so if the list changes, *both* variables refer to the changed list. If you want to make a copy instead, there are several ways to do it. Here are two:

In [14]:
```
list3 = list1[:]
list4 = list1.copy()
print (list1)
print (list3)
print (list4)

list1.append(8)
print (list1)
print (list3)
print (list4)
```

```
[0, 0, 0, 0, 2, 7]
[0, 0, 0, 0, 2, 7]
[0, 0, 0, 0, 2, 7]
[0, 0, 0, 0, 2, 7, 8]
[0, 0, 0, 0, 2, 7]
[0, 0, 0, 0, 2, 7]
```

That first line is a little interesting. It is actually a special case of *list slicing*: copying specified indices out of a list. The syntax of slicing is:

```
newlist = oldlist[l:h]
```

which *copies* the indices `[l:h])` of the old list (note that the interval on the right is open) to the new list.

In [15]:
```
orig = [3, 1, 7, 3, 9, 2]
slice1 = orig[2:5]
slice2 = orig[:4]
slice3 = orig[3:]
print(slice1)
print(slice2)
print(slice3)
```

```
[7, 3, 9]
[3, 1, 7, 3]
[3, 9, 2]
```

Note that if you do not provide `l`, it defaults to 0, and if you do not provide `h`, it defaults to `len(list)`

You can also add an optional third argument that specified a stride, to, for example, copy every 2nd element of the list:

In [16]:
```
slice4 = orig[0:len(orig):2]
print(slice4)
```

```
[3, 7, 9]
```

You can even use a negative stride to walk over the list in reverse (note that `l` and `h` will change places in that case):

In [17]:
```
slice5 = orig[::-1]
print(slice5)
```

```
[2, 9, 3, 7, 1, 3]
```

Finally, note that lists can contain other lists:

In [18]:
```
nested = [[0, 1], [2, 3], [4, 5, 6]]
print(nested)
```

```
[[0, 1], [2, 3], [4, 5, 6]]
```

Strings

Interestingly, lists are not the only sequence type we have encountered. Strings in Python are also sequence types:

In [19]:
```
string1 = 'Hello'
print(string1)
```
```
Hello
```

In [20]:
```
print(len(string1))
for s in string1 :
    print (s)
```
```
5
H
e
l
l
o
```

Tuples

Tuples in Python are kind of like lists, except that unlike lists, you cannot change them once they are created: you cannot make them longer, remove elements, or even change the elements themselves:

In [21]:
```
tuple1 = (1.5, 2.7)
print(tuple1)
```
```
(1.5, 2.7)
```

In [22]:
```
print(tuple1[0])
```
```
1.5
```

In [23]:
```
print(tuple1[1])
```
```
2.7
```

In [24]:
```
for i in tuple1 :
    print (i)
```
```
1.5
2.7
```

```
In [25]: tuple1.append('x')
```

```
---------------------------------------------------------------------------
----
AttributeError                            Traceback (most recent call l
ast)
<ipython-input-25-6773d97c83ad> in <module>
----> 1 tuple1.append('x')

AttributeError: 'tuple' object has no attribute 'append'
```

```
In [26]: tuple1[0] = 3
```

```
---------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
<ipython-input-26-5e0f22de5ab3> in <module>
----> 1 tuple1[0] = 3

TypeError: 'tuple' object does not support item assignment
```

One thing to note, though, is that if an element of a tuple is a *reference to some other thing*, you can still change that other thing -- you just can't change the tuple itself. So, for example, if a tuple has a list as one of its elements, the list can be changed, you just can't make the tuple refer to a different list.

```
In [27]: tuple2 = (2.3, [])
```

```
In [28]: print(tuple2[1])
```

```
[]
```

```
In [29]: tuple2[1].append(2)
```

```
In [30]: print(tuple2[1])
```

```
[2]
```

```
In [31]: print(tuple2)
```

```
(2.3, [2])
```

```
In [32]: tuple1
```

```
Out[32]: (1.5, 2.7)
```

One useful way of extracting things out a tuple is *unpacking* them, using the following notation. (Note that this is what is happening under the hood if you write a function that returns multiple values: those values are packaged up into a tuple, that is then unpacked)

In [33]:
```
i, j = tuple1
```

In [34]:
```
print(i)
```

```
1.5
```

In [35]:
```
print(j)
```

```
2.7
```

Sets

Sets are data structures with the following properties:

1. They are *unordered*: the order that you retrieve elements from a set (if you are printing them out or iterating over them) is not guaranteed.
2. They are *unique*: any element can only exist in the set once.

In [36]:
```
set1 = {'a', 'b', 'c'}
print(set1)
```

```
{'c', 'b', 'a'}
```

In [37]:
```
set2 = {'a', 'b', 'c', 'a'}
print(set2)
```

```
{'c', 'b', 'a'}
```

In [38]:
```
for s in set1 :
    print(s)
```

```
c
b
a
```

In [39]:
```
set2.add('d')
print(set2)
```

```
{'c', 'b', 'd', 'a'}
```

In [40]:
```
set2.remove('a')
print(set2)
```

```
{'c', 'b', 'd'}
```

In [41]:
```
set3 = set()
print(set3)
```
```
set()
```

In [42]:
```
set3.add('a')
print(set3)
```
```
{'a'}
```

In [43]:
```
set3 = set()
set3.add('a')
set3.add('a')
print(set3)
```
```
{'a'}
```

Dictionaries

Dictionaries are like sets, except instead of just holding individual items, they hold *pairs* of items: a *key* and a *value*. Each key is associated with a value, and a dictionary guarantees that any key appears in the dictionary at most once:

In [44]:
```
dict1 = {'a' : 0, 'b' : 1, 'c' : 3}
print(dict1)
```
```
{'a': 0, 'b': 1, 'c': 3}
```

In [45]:
```
print(dict1['a'])
```
```
0
```

In [46]:
```
print(dict1['b'])
```
```
1
```

In [47]:
```
print(dict1['c'])
```
```
3
```

In [48]:
```
print(dict1['d'])
```
```
-----------------------------------------------------------------------
----
KeyError                                  Traceback (most recent call l
ast)
<ipython-input-48-4c418ccf3f33> in <module>
----> 1 print(dict1['d'])

KeyError: 'd'
```

In [49]:
```
dict1['d'] = 4
print(dict1['d'])
```

```
4
```

In [50]:
```
dict1['a'] = 3
print(dict1['a'])
```

```
3
```

In [51]:
```
for k in dict1 :
    print (k)
    print (dict1[k])
```

```
a
3
b
1
c
3
d
4
```

In [52]:
```
for k, v in dict1.items() :
    print("Key {} has value {}".format(k, v))
```

```
Key a has value 3
Key b has value 1
Key c has value 3
Key d has value 4
```

In [53]:
```
len(dict1)
```

Out[53]:
```
4
```

In [ ]:
```

```