**Lecture 10 (week 6.1)**

Tuesday October 11th 2016

This lecture introduces the concept of Python containers. Containers are any object that holds an arbitrary number of other objects. Generally, containers provide a way to access the contained objects and to iterate over them. Different containers store data differently and allow their access in different ways. Some containers are more convenient than others for some specific tasks.

Today we will focus on **lists** and **tuples**.

Containers

These are the containers we will study in class (some of them we have seen already):

* **lists** – the most popular container data type in python; can store any number of any objects.
* **tuples** – similar to list, yet once created are immutable,
* **sets** – can store only unique elements,
* **dictionary** – also known as associative arrays. They contain mapping of keys into value,
* **str** – string, a sequence of unicode characters,

Containers in Python can be either **mutable** or **immutable**. The fact that a container object is immutable doesn’t always mean that the objects it holds are also immutable (e.g. an immutable tuple holding mutable lists).

Examples of *mutable containers* include:

* list, set, dictionary,

Examples of *immutable containers* include:

* string, tuple.

Container object can store their content in either an **ordered** or **unordered** manner. Order, or lack of thereof, is unrelated to the mutability of objects.  This means that both mutable and immutable objects can be either ordered or unordered.

Examples of *ordered containers* include:

* list, string, tuple

Examples of *unordered containers* include:

* dictionary, set

From the list above, we will look at lists, tuples, dictionary in detail, you already know about strings.

Lists

Lists are the most versatile of Python's containers. A list contains items separated by commas and enclosed within square brackets **[]**. All the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator **[]** and **[:]**.

The plus **+** sign is the list concatenation operator, and the asterisk **\*** is the repetition operator.

Read sections 10.1 to 10.13 from <http://greenteapress.com/thinkpython2/html/thinkpython2011.html>

**In [1]: a = [1,2,4,8,16]** *# list of ints*

**In [2]: a**

**Out[2]: [1,2,4,8,16]**

**In [3]: c = [4,'candles',4.0,'handles']** *# can mix types*

**In [4]: c[2] = 'knife'**

**In [5]: c[-1]** *# negative indices count from end*

**Out[5]: 'handles'**

**In [6]: c[1:3]** *# slicing*

**Out[6]: ['candles', 'knife']**

**In [7]: c[2:]** *# omitting defaults to start or end*

**Out[7]: ['knife','handles']**

**In [8]: c[0:4:2]** *# variable stride (could just write c[::2])*

**Out[8]: [4,'knife']**

**In [9]: a + c** *# concatenate*

**Out[9]: [1,2,4,8,16,4,'candles','knife','handles']**

The examples below shows some alternative ways to define lists:

**In [12]: L = []** *#empty list*

**In [14]: L = list()** *#empty list*

**In [16]: A = B = []** *# both names will point to the same list*

**In [17]: A = []**

**In [18]: B = A** *# both names will point to the same list*

**In [19]: A = []; B = []** *# independent lists*

There are *methods* specific to lists, here they are:

[list.append(obj)](http://www.tutorialspoint.com/python/list_append.htm)  
Appends object obj to list

[list.count(obj)](http://www.tutorialspoint.com/python/list_count.htm)  
Returns count of how many times obj occurs in list

[list.extend(seq)](http://www.tutorialspoint.com/python/list_extend.htm)  
Appends the contents of seq to list

[list.index(obj)](http://www.tutorialspoint.com/python/list_index.htm)  
Returns the lowest index in list that obj appears

[list.insert(index, obj)](http://www.tutorialspoint.com/python/list_insert.htm)  
Inserts object obj into list at offset index

[list.pop(obj=list[-1])](http://www.tutorialspoint.com/python/list_pop.htm)  
Removes and returns last object or obj from list

[list.remove(obj)](http://www.tutorialspoint.com/python/list_remove.htm)  
Removes object obj from list

[list.reverse()](http://www.tutorialspoint.com/python/list_reverse.htm)  
Reverses objects of list in place

[list.sort([func])](http://www.tutorialspoint.com/python/list_sort.htm)  
Sorts objects of list, use compare func if given

Here is an example how to use some of these list methods:

**In [45]: L=['physics', 'chemistry', 1997, 2000]**

**In [47]: L.append('object')**

**In [48]: L**

**Out[48]: ['physics', 'chemistry', 1997, 2000, 'object']**

**In [49]: name=L.pop()**

**In [50]: L**

**Out[50]: ['physics', 'chemistry', 1997, 2000]**

**In [51]: name**

**Out[51]: 'object'**

I would like to emphasize the concept of "in place" operations versus 'new object' creation:

**In [18]: a = [1,2,3]**

**In [19]: b = [4,5,6]**

**In [20]: c = a + b** *# concatenating with creating a new list*

**In [21]: c**

**Out[21]: [1, 2, 3, 4, 5, 6]**

**In [22]: a**

**Out[22]: [1, 2, 3]**

**In [23]: a.extend(b)** *# append b to a (in place modification of a)*

**In [24]: a**

**Out[24]: [1, 2, 3, 4, 5, 6]**

This concept of "in place" versus "new object" has also some meaning with functions, for functions that return a new object and function that return a new object AND change the values of the input object. I will say more about this in some future lecture, for now you don’t have to worry too much about this, although you may face this problem occasionally, for instance:

**In [37]: a**

**Out[37]: [3, 2, 1]**

**In [38]: b=a.sort()**

**In [39]: type(b)**

**Out[39]: NoneType**

**In [40]: a**

**Out[40]: [1, 2, 3]**

The list method **sort()** performs an in place operation, and the function does not return anything (it is **NoneType**).

Some Python keywords (**<http://www.programiz.com/python-programming/keyword-list>**) word on lists, e.g. **del:**

**In [7]: list1 = ['physics', 'chemistry', 1997, 2000]**

**In [9]: print(list1)**

**['physics', 'chemistry', 1997, 2000]**

**In [10]: del list1[2]**

**In [11]: list1**

**Out[11]: ['physics', 'chemistry', 2000]**

Note that many of the build-in functions (<https://docs.python.org/2/library/functions.html#>) work on lists, e.g. **len()**, **max()**, **min(), cmp()** and many others (experiment yourself!). For instance, the **enumerate()** build-in function (<https://docs.python.org/2/library/functions.html#enumerate>) allows you to iterate among the elements of a list:

**In [1]: L=['physics', 'chemistry', 1997, 2000]**

**In [7]: for index,item in enumerate(L):**

**...: print(index,item)**

**...:**

**0 physics**

**1 chemistry**

**2 1997**

**3 2000**

It is possible to define a list of lists. To access a list of list of list elements, you use the following notation:

**In [1]: a=[3,4,[1.2,'g'],'aa']**

**In [2]: type(a)**

**Out[2]: list**

**In [3]: a[0]**

**Out[3]: 3**

**In [4]: a[2]**

**Out[4]: [1.2, 'g']**

**In [5]: a[2][0]**

**Out[5]: 1.2**

**In [6]: a[2][1]**

**Out[6]: 'g'**

**In [7]: type(a[2])**

**Out[7]: list**

Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses. The main differences between lists and tuples are: Lists are enclosed in brackets **[ ]** , and their elements and size can be changed, while tuples are enclosed in parentheses **( )** and cannot be changed. Tuples can be thought of as **read-only** lists, they are **immutable**.

Below are a list of tuples definitions and operations which covers the essentials:

**In [11]: q = (1, 2, 4, 8, 16)** *# tuple of integers*

**In [12]: q**

**Out[12]: (1, 2, 4, 8, 16)**

**In [13]: r = (4, 'candles', 4.0, 'handles')** *# can mix types*

**In [14]: s = ('lonely',)** *# singleton*

**In [15]: t = ()**  *# empty*

**In [16]: r[1]**

**Out[16]: 'candles'**

**In [17]: r[1]='knife'** *# cannot change tuples*

**---------------------------------------------------------------------------**

**TypeError Traceback (most recent call last)**

**<ipython-input-17-1e153d5ec383> in <module>()**

**----> 1 r[1]='knife' # cannot change tuples**

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

**In [18]: u = 3, 2, 1**  *# parentheses not necessary*

**In [19]: v, w = 'this', 'that'**

**In [20]: v**

**Out[20]: 'this'**

**In [21]: w**

**Out[21]: 'that'**

**In [22]: type(v)**

**Out[22]: str**

**In [24]: tt=v,w**

**In [25]: type(tt)**

**Out[25]: tuple**

Note that we have already seen several examples of functions returning *tuples*, although it was not *named* like this. This was the case for the **scipy.optimize.curve\_fit()** and the **matplotlib.pyplot.hist()** methods. In general a tuple is a very convenient way for a function to return more than one variable and variables of different type, for instance the following function takes a numerical value and returns the value and its string version in a tuple:

**In [30]: def floatstring(x):**

**....: return x,str(x)** *# use of parenthesis is optional*

**....:**

**In [31]: tt=floatstring(4.5)**

**Out[31]: (4.5, '4.5')**

**In [33]: type(tt)**

**Out[33]: tuple**

**In [34]: type(tt[0])**

**Out[34]: float**

**In [35]: type(tt[1])**

**Out[35]: str**

Tuple methods are **count()** and **index()** (I leave it to you to experiment with them).

Note that, for defining a tuple, the comma "**,"** is mandatory and parenthesis **()** is not:

**In [41]: s=('lonely')**

**In [42]: type(s)**

**Out[42]: str**

**In [43]: s=('lonely',)**

**In [44]: type(s)**

**Out[44]: tuple**

**In [45]: s='lonely',**

**In [46]: type(s)**

**Out[46]: tuple**

You can now read the sections in <http://greenteapress.com/thinkpython2/html/thinkpython2013.html>