**Lecture 7 (week 4.2)**

Thursday 29th 2016

Loops and iterations

The last fundamental concept in programming we have to cover is *iterations*. For many situations, a program has to do a certain task repetitively; it is then convenient to do it through *iterative loops*. The **for** keyword in Python allows you to iterate through a countable list of elements inside what is called an *iterable object.* We have already seen examples of iterable objects: arrays and strings. Python has other forms of iterable objects, such as *Python lists* (which I will introduce today). So an *iterable* list of elements could take various forms, and today we will cover the most common situations: the case of strings, Python lists and **range()**.

Note: It is usually possible to replace a **for** block by a **while** block, although the **for** block could be a lot more convenient in some situations, especially when the iteration does not need a conditional statement and only consists in enumerating single elements from an ensemble of elements.

For loop on a string

When the iterable is a string, the **for** loop allows you to iterate over all individual characters. The syntax is

**Some previous statements from main code**

**for char in string :**

**Some statements using the variable char**

**Some more statements from main code**

The syntax of the **for** loop is straightforward:

**In [2]: mystring=‘somestuff'**

**In [3]: for c in mystring:**

**...: print("character ",c)**

**...:**

**character s**

**character o**

**character m**

**character e**

**character s**

**character t**

**character u**

**character f**

**character f**

Note that the variable **c** could have any name, and it is a string. Python creates this variable as soon as it sees the **for c** statements. This means that **c** is **str** type.

For loop on a list

Let me first give a very short introduction to the concept of *lists* in Python. A list is a sequence of variables which can be of any type. The elements of a list can be accessed individually (like for an array). The most common way to create a Python lists is to use the square brackets **[……]**:

**In [1]: import math**

**In [2]: mylist=[2,math.pi,math.sqrt(3.),'somestuff']**

**In [3]: type(mylist)**

**Out[3]: list**

**In [4]: mylist**

**Out[4]: [2, 3.141592653589793, 1.7320508075688772, 'somestuff']**

There is a lot more to say about lists, which I won’t cover today. All we need to know is that the **for** statement allows you to enumerate the elements of a list:

**In [5]: for v in mylist:**

**...: print(type(v),"v= ",v)**

**...:**

**<class 'int'> v= 2**

**<class 'float'> v= 3.141592653589793**

**<class 'float'> v= 1.7320508075688772**

**<class 'str'> v= somestuff**

Like in the case of strings, you are free to name the enumeration variable (I used **v** in the example above). As you see from the example, Python is updating the variable **v** *attributes* and *type* every time it picks up a new element from the list.

A variant is to access the list elements with the conditional **while** statement we explored las time:

**In [28]: i=0**

**In [29]: imax=len(mylist)**

**In [30]: while i < imax:**

**....: print(i,mylist[i])**

**....: i=i+1**

**....:**

**0 2**

**1 3.141592653589793**

**2 1.7320508075688772**

**3 somestuff**

As an exercise, give some thoughts as to why I used **while i < imax:** and not **while i <= imax:**

For loop with **range()**

There is a convenient functionality in Python, **range(n)**, which creates an iterable object containing the sequence of integers from 0 to n-1:

**In [32]: ll=range(5)**

**In [33]: ll**

**Out[33]: range(0, 5)**

**In [34]: type(ll)**

**Out[34]: range**

Note that the type is **range**, so is it a list or not? It “looks” like a list, but Python treats **range(n)** slightly differently than a list internally: in fact everything is the same as a list for calculations, but a **range()** variable does not create numbers stored in the memory, instead Python just “knows” that this is a list of iterable integers from 0 to n-1. This is the only difference, but for big calculations, it will result in a computational speed gain that the list does not occupy some space in memory, i.e. it does not have to be accessed physically. You can access range elements like a list:

**In [37]: for i in ll:**

**....: print(i)**

**....:**

**0**

**1**

**2**

**3**

**4**

Typically **range(n)** is useful when you want to access sequentially the elements of a list without creating a variable each time it reads elements, this will result in a computing speed gain for big calculations.

**In [42]: import math**

**In [43]: import numpy as np**

**In [44]: mylist=[2,math.pi,math.sqrt(3.),'somestuff',np.array(2)]**

**In [45]: ll=range(5)**

**In [46]: for i in ll:**

**....: print(i,mylist[i])**

**....:**

**0 2**

**1 3.141592653589793**

**2 1.7320508075688772**

**3 somestuff**

**4 2**

Note that with the command **list()** you can transform an iterable object into a ‘real Python list':

**In [1]: a=range(5)**

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

**range(0, 5)**

**In [3]: b=list(a)** *# now b is a list!*

**In [4]: print(b)**

**[0, 1, 2, 3, 4]**

**In [5]: type(a)**

**Out[5]: range**

**In [6]: type(b)**

**Out[6]: list**

Note that **range()** can take up to three arguments, the third (optional) is the step between elements (see <http://pythoncentral.io/pythons-range-function-explained/>) for a full description).

You can now read Sections 7.1 to 7.8 in <http://greenteapress.com/thinkpython2/html/thinkpython2008.html> for a summary on the iteration procedures in Pyhton and Section 8.1 to 8.10 from <http://greenteapress.com/thinkpython2/html/thinkpython2009.html>

The conditional statements can be used for control flow in Python, along with other control flow tools:

<https://docs.python.org/3/tutorial/controlflow.html>