Lecture 2

Ashwini Vaidya

March 20, 2017

0.1 Dynamic Typing

Python uses dynamic typing, which means that it is not necessary to declare the type of a variable during assignment. In fact, variables themselves do not belong to a type, but they reference an object, which is is typed (string, integer, list etc.). When a variable is re-assigned to a string, integer or list, internally its reference will be updated to a new object. The old reference to the object is 'garbage collected' such that it no longer needs to stay in memory (See Ascher and Lutz, Ch 6 for more details).

0.2 Lists

Lists are objects that are *mutable*, unlike strings. This means that lists can be changed in-place. Elements inside a list may be added or deleted in place

Some useful properties of lists are that they are ordered from left to right as they are treated as *sequences* (like strings). They are also nestable, i.e. a list can contain another list as well as extendable, i.e. their length may be increased or decreased. The slicing syntax that is applicable for strings is also used for lists.

We have already seen that lists can also be concatenated and multiplied like strings.

```
>>> [1,2,3] +[4,5,6]
[1, 2, 3, 4, 5, 6]
>>>
>>> ['gollum'] * 3
['gollum', 'gollum', 'gollum']
```

Slicing works in the same way, except that a given index returns the object inside a list, whereas a [start:end] slice will return a new list:

```
>> l=[‘Already’,‘found’,‘a’,‘way’]
>>> l[0]
‘Already’
>>> l[1:3]
[‘found’, ‘a’]
```

As we saw in the homework assignment, slicing also enables us to change a list in place, by assigning to a particular slice. We can't do this with strings as they are immutable. Assignment using slicing syntax first deletes an element and then inserts the new one.

```
>>> l=‘gollum’
>>> l[4:6] =‘y’
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ’str’ object does not support item assignment

>>> l=[1,2,3]
>>> l[1:2] =[9,9]
>>> l
[1, 9, 9, 3]
>>
```

0.2.1 List methods

Slicing is not the oft-used way of adding and removing items from a list. While it's often used to reference objects that are part of a list, list methods are a preferred way of adding and deleting elements. The most commonly used methods are **append**, **extend** and **pop**. Another is **sort** and **reverse**.

The **append** method will take exactly one item (object) and add it to the end of a list.

```
>>> l=[‘Already’,‘found’,‘a’,‘way’]
>>> l.append(‘today’)
>>> l
[‘Already’, ‘found’, ‘a’, ‘way’, ‘today’]

>>> l.append(’today’,’yeah’)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: append() takes exactly one argument (2 given)
```

If you wish to join together more than one item to a list (e.g. another list), it is better to use **extend**. If you use **append**, the results will not be as you expect, you will get a nested list instead:-

```
>>> l=['Already','found','a','way']
>>> l.extend(['today','too'])
>>> l
['Already', 'found', 'a', 'way', 'today', 'too']

>>> l=['Already','found','a','way']
>>> l.append(['today','too'])
>>> l
['Already', 'found','a', 'way', ['today', 'too']]
```

0.3 Truth Tests using if ..

An `if ..` statement is a conditional and will select an action to be performed on the basis of some truth value. It's a way of formulating a logical test in Python. However, it also changes the control flow of the program, causing it to jump or skip parts of program.

`if` statements will consist of a header line terminated by a colon, with an indented block (or blocks) below it. (We have discussed the syntax of indented lines before in the simple_words.py program). In the interpreter, an `if` statement automatically brings up continuation dots on the next line (expecting an indented block to follow). When an unindented blank line is found, the interpreter assumes that the statement is complete and runs the entire block.

```
>>> x = 'python'
>>> if x=='python':
...     print 'monty'
...
monty
```

`if` is used to check for comparison, equality and truth values.

The == operator test will allow for equivalence of objects (this is not to be confused with =, which is used for assignment only–very different things!

The **is** operator checks for identity of objects

```
>>> l1=[1,2,3]
>>> l2=[1,2,3]
>>> l1==l2
True
>>> l1 is l2
False

>>> s1='this is a string'
>>> s2='this is a string'
>>> s1==s2
True
>>> s1 is s2
False
```

l1 and l2 (and s1 and s2) are equivalent i.e. their components are exactly the same. However, they are not the same object in memory and hence fail the *identity* test.

The result 'True' and 'False' are Boolean truth values, where True stands for the integer '1' and the False. In general, numbers are true if they're non-zero and other objects are true if they are not empty.

Object	Value
"hello"	True
""	False
[]	False
{}	False
1	True
0.0	False
None	False

Table 1: Truth values (from Ascher and Lutz, pg 188)

Very often, methods on objects such as string methods will return a truth value which you can use to carry out certain tasks. For instance, **isupper()** returns a truth value:

```
>>> 'U'.isupper()
True
>>> 'U'.islower()
False
>>> 'Maria'.endswith('a')
```

```
True
>>> 'Maria'.endswith('ia')
True
```

References

Learning Python by Ascher and Lutz