A Crash Course in Basic Python Commands with Sage

1 Comments

The symbol # at the beginning of a line means that it is a line. The triple double quotes """ are reserved for comments along several lines. If you know C/C++, they are equivalent to // and /*...*/.

```
1 # Elements of Python Programming Language
2 """
3 more comments
4 more lines
5 """
```

Of course, do not expect any output from this program.

The command `print` complete it with the customary salute.

```
1 # Elements of Python Programming Language
2 """
3 more comments
4 more lines
5 """
6 print 'Hello World!'
```

2 Types

According to the manuals, Python is strongly-typed. If you know what this jargon means, the following program will probably amaze you:

```
1 #############
2 #TYPES
3 #############
4 a = 2
5 print 3*a
6 a = 'Hi'
7 print 3*a
```

The output is

```
6
HiHiHi
```

Inserting `print type(a)` after lines 5 and 7, you get `<type 'sage.rings.integer.Integer'>` in the first case and `<type 'str'>` in the second. The first variable is an integer and the second a string

3 Data structures

The basic data structures built-in in python are lists, tuples and dictionaries.

Lists. A list is in principle a one-dimensional array like in many other programming languages. It is really a list of elements separated by commas and between brackets `[...,...,...]`. Its elements are called inicated its order between brackects, for instance `L[0]` means the first element of the list, `L[1]` is the second element of the list and so on. Like in C/C++ the computers starts counting from zero. The main difference with other languages is that the elements of a list can be objects of different nature (type). For instance you can combine numbers, string and even lists.

The lists also admit slicing like for instance in Matlab or Octave and `L[n:m]` means the elements from `L[n]` to `L[m-1]`. If n or m (or both) are omitted the starting or the finishing point are the beginning or the end of the list. Negative values are identified with

All of these consideration should be clear with the following code

```
1  ##################
2  #DATA STRUCTURES
3  ##################
4  # lists
5  L = [1,2,'three',4,'five']
6  print L
7  print L[3]
8  print L[2:4] # L[2] included, L[4] not included
9  print L[2:] # From L[2]
10 print L[:-1] # To the last but one element
```

giving the output

```
[1, 2, 'three', 4, 'five']
4
['three', 4]
['three', 4, 'five']
[1, 2, 'three', 4]
```

Especially important lists are finite arithmetic progressions. They are created with `range`(n, m, s) where n is the starting point, s the step and m a strict upper limit. If the step is omitted it is assumed $s = 1$. If besides n is omitted it is assumed $n = 0$. For example

```
1 # special lists
2 L = range(20) # from 0 to 20 (not included)
3 print L
4 L = range(5,20,2) # Same thing starting from 5 and step=2
5 print L
6 L = range(5,13) # Integers in [5,13)
7 print L
```

gives

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[5, 7, 9, 11, 13, 15, 17, 19]
[5, 6, 7, 8, 9, 10, 11, 12]
```

Tuples. Tuples syntax is similar to list syntax changing brackets by parenthesis in the definition. Tuples are immutable lists. This means that they behave in the same way but you cannot redefine their elements. The following program leads to an error uncommenting the last but one line.

```
# tuples vs lists
L = [1,2,'three',4,'five']
print L[2:4]
L[2] = 'trois'
print L
T = (1,2,'three',4,'five')
print T[2:4]
#T[2] = 'trois'
print T
```

Strings and tuples are alike.

```
salute = "how are you doing?"
print salute[2:6]
print salute[:-1]
salute[2]='z'
```

gives the expected answer for the first lines and the expected error message for the last

```
w ar
how are you doing
--------------------------------------------------------------------------
                  [omitted error messages]
TypeError: 'str' object does not support item assignment
```

Dictionaries. Dictionaries are hash tables. A list of names and their meaning. The syntax is a list of `name: meaning` between curly brackets.

```
# dictionaries
D = {'one': 1, 'two': 'deux', 'e': '2.718...', 'three': 'trois'}
print D
print 'Note that there is not a determined order in a dictionary '
print D['two']
print D['e']
print 7*D['one']
```

```
{'three': 'trois', 'e': '2.718...', 'two': 'deux', 'one': 1}
Note that there is not a determined order in a dictionary
deux
2.718...
7
```

4 Flow control statements

The flow control statements in Python are `for`, `if` (`if-else`) and `while`. The most original convention is that block indentation is mandatory but easy. When you mark the end of a flow control statement when a colon `:` the editor automatically force you to indent.

In flow control statements the keyword `in` appears very often to indicate elements in a list (or tuple or string).

Let us start with a very easy example

```
1  ##################
2  #FLOW CONTROL
3  ##################
4  for i in range(10):
5      print i,
6  print
7  for i in range(3):
8      print i,'->',i^2
```

that allows us also to learn that a comma , after print avoid a change of line

```
0 1 2 3 4 5 6 7 8 9
0 -> 0
1 -> 1
2 -> 4
```

The `if` statement and `if-else` blocks work like in other languages

```
1  friends = ['John','Bob','George','Henry','Alice']
2  print friends
3  for name in friends:
4      print name,
5      if name == 'George':
6          print '(My best friend) ,',
7      elif name == 'Alice':
8          print ''
9      else:
10         print ', ',
```

The first `if` detects our best friend and the second, disguised as `elif` (else if), omits the last comma.

```
['John', 'Bob', 'George', 'Henry', 'Alice']
John ,  Bob ,  George (My best friend) , Henry ,  Alice
```

Finally, we illustrate `while` statement with program comparing pythonic and non-pythonic (despective term applied when one tries to write Python code using other languages philosophy)

```
1  sentence = 'The last example'
2  # Non Pythonic
3  i=0
4  while i <len(sentence):
5      print sentence[i],
6      i+=1 # abbreviation of i=i+1
7  print ''
8  # Pythonic
9  for i in sentence:
10         print i,
11 print ''
```

The output is the same in both cases. Note that (like in C/C++) the abbreviation `i+=k` is available (but `++i` is not). The command `len` indicates the length of the string.

```
T h e   l a s t   e x a m p l e
T h e   l a s t   e x a m p l e
```