Python Crash Course

Darin Brezeale

December 13, 2011

This is a short introduction to python that assumes you are already familiar with programming in C. I just cover the basics; there are many longer tutorials available. Python is a high-level, object-oriented, interpreted language. We can process python code by either typing commands directly into the interpreter or by using the interpreter to process a file of commands (what I would call an actual program). Most python tutorials and books will demonstrate python commands by typing them into the interpreter; you will recognize this by the `>>>` prompt. I'll do some of this as well, but much of the code shown is from a separate source file.

1 Starting the Python Interpreter

Assuming that the python interpreter is in your path, typing `python` will result in something like the following:

```
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the `>>>` prompt, you can type python commands. To exit from the python interpreter, press `ctrl-d`.

2 Variables

To create variables, we just use them:

```
>>> x = 4
>>> y = 8.0
>>> x*y
32.0
```

Notice that I never declared the variable types. Python will make assumptions about the variable types. We can see the variable types using the `type` command:

```
>>> type(x)
<type 'int'>
>>> type(y)
<type 'float'>
```

Variables are dynamically typed, so we can change the type like this:

```
>>> z = 3
>>> type(z)
<type 'int'>
>>> z = 3.0
>>> type(z)
<type 'float'>
```

3 Strings

String variables can be created by enclosing the string in single or double quotes. If you wish for the string to cover several lines, then enclose it in sets of three quotes.

```
first = 'Bob'
last = "Jones"
longString = """this string may be too
long to fit on a single line"""

print "%s %s was here" % (first, last)
print "the first character of %s is %c" % (first, first[0])

whole = first + last    # string concatenation

print whole
print longString
```

produces

```
Bob Jones was here
the first character of Bob is B
BobJones
this string may be too
long to fit on a single line
```

This example shows several things:

1. Comments can be created using the pound sign (i.e., #).
2. Strings are easily concatenated.
3. Individual characters within a string can be accessed.
4. The long string is printed as-is with the line break.
5. The `print` statement can be used in a variety of ways. Note that in python 3 the print statement will be a function, that is, you will use something like `print(some_string)`.

4 Loops

Python supports `while` and `for` loops.

```
x = 5
while x < 10:
    print x
    x += 1
```

There are a couple of differences between this code and the equivalent C code:

1. The indentation is not just for clarity; it's required. Whereas in C and some other languages the body of the `while` loop would be enclosed in curly braces, in python it is designated by indenting the code. This is also true for `for` loops, conditionals, functions, and classes.

2. Parentheses are not required for the test condition. However, the line with the test condition is terminated by a colon.

We can use a `for` loop to iterate through a set of values:

```
for i in range(5):
    print i
```

produces

```
0
1
2
3
4
```

In this case the `range` function generates the sequence $0, 1, \ldots, 4$.

It's also possible to specify the range and an integer step size:

```
from math import sqrt

for i in range(20,27,2):
    print "the square root of %d is %5.3f" % (i, sqrt(i))
```

which produces

```
the square root of 20 is 4.472
the square root of 22 is 4.690
the square root of 24 is 4.899
the square root of 26 is 5.099
```

The values here range from 20 to 26 (i.e., 27-1) in steps of 2. The `sqrt` function is in the `math` module, so we need to import it to use it.

As we will see later, the `for` loop is more versatile than what you are used to.

5 Conditionals

Conditional operation is performed in python using a combination of `if`, `elif`, and `else` statements; there is no `switch` statement.

```
x = 5
if x == 4:
    print 'x is 4'
elif x < 10:
    z = 5
    if x == 5:
        print 'x is less than 10'
        print "z is %d" % z
```

As stated above, the indentation is required and matches the flow of the code, so this code produces

```
x is less than 10
z is 5
```

6 Data Structures

Python has several built-in data structures, in particular `lists`, `dictionaries`, `tuples`, and `sets`. Of these, I will only discuss the first two. A list is used much like one would use an array in other languages.

```
someList = [4, 5, 6, 23, 14]
namesList = ['Bob', 'John', 'Mary']

print someList
print someList[0]

print namesList
print namesList[2]
```

produces

```
[4, 5, 6, 23, 14]
4
['Bob', 'John', 'Mary']
Mary
```

Note that the first element of the list has an index value of 0. One big difference between lists in python and arrays in C is that we can mix data types. The following list consists of an integer, a floating point number, and a string:

```
someList = [4, 17.345, 'Jack']

print someList
print someList[1]
```

produces

```
[4, 17.344999999999999, 'Jack']
17.345
```

We can also work with parts of a list by *slicing* it:

```
someList = [4, 17.345, 'Jack', 89, 'Bob']

print someList[1:3]
print someList[2:]
```

produces

```
[17.344999999999999, 'Jack']
['Jack', 89, 'Bob']
```

The first `print` statement prints elements with indices of 1 through 2 (i.e., 3-1). The second `print` statement prints the elements from index 2 until the end of the list.

The previously mentioned versatility of the `for` loop lets us step through a list:

```
someList = [4, 7, -5, 9]

for i in someList:
    print i*i
```

produces

```
16
49
25
81
```

List comprehension allows us to use a `for` loop to produce a list:

```
someList = [4, 7, -5, 9]

newList = [i*i for i in someList]

print newList
```

produces

```
[16, 49, 25, 81]
```

Once we have a list, we may wish to modify it.

```
data = [5, 'Darin', 8]

data.append("John")
print data

data.insert(2, 44)    # insert 44 as the third element
print data

data[1] = 75    # change the element with an index of 1
print data

print "data has %d objects" % len(data)
```

produces

```
[5, 'Darin', 8, 'John']
[5, 'Darin', 44, 8, 'John']
[5, 75, 44, 8, 'John']
data has 5 objects
```

The dictionary data structure is a hash table. It consists of key:value pairs.

```
someDict = {"John": 45, "Bob": 73}

print someDict["Bob"]

print "the keys of someDict are "
print someDict.keys()

print someDict

del someDict["Bob"]    # delete the object with Bob as the key
print someDict

someDict["Mary"] = 5    # add an object with a key of Mary
print someDict
```

produces

```
73
the keys of someDict are
['Bob', 'John']
{'Bob': 73, 'John': 45}
{'John': 45}
{'John': 45, 'Mary': 54}
```

7 Functions

The `def` keyword is used to create a function.

```
def cube(a):
    return a**3

x = 5
y = cube(x)
print "%d cubed is %d" % (x, y)
```

produces

```
5 cubed is 125
```

We can also return multiple values from a function:

```
def almost(aList):
    """return second smallest
       and largest values"""
    aList.sort()    # sort the list in place
    size = len(aList)
    smallest = aList[1]
    largest = aList[size - 2]

    return (smallest, largest)


data = [21,3,8,7,34,2,19]

first, last = almost(data)

print "the second smallest and largest values",
print "are %d and %d" % (first, last)
```

produces

```
the second smallest and largest values are 3 and 21
```

This program also demonstrates that comments that span multiple lines can be enclosed in sets of triple double quotes.

8 File I/O

Let's say we have the file `data.csv`, which has the following contents:

```
4,5,6
18,19,20
42,43,44
35,36,37
```

There are several ways to read this file; here's one:

```
fp = open("data.csv")
dataList = fp.readlines()    #  read a file, storing each line as a string
fp.close()

print dataList
```

produces

```
['4,5,6\n', '18,19,20\n', '42,43,44\n', '35,36,37\n']
```

Let's say that we wish to sum the values on each line. To do so, we can tokenize each line and then convert each of the tokens to integers:

```
fp = open("data.csv")
dataList = fp.readlines()    #  read a file, storing each line as a string
fp.close()

for line in dataList :
    t = line.split(',')      # produce list of tokens; comma is delimiter
    mysum = 0
    for token in t :
        mysum = mysum + int(token)  # tokens are strings, so convert
    print mysum
```

We can also write to files:

```
fp = open("output.csv", "w")         # open a file for writing

outString = ""                       # create empty string
i = 1
while i <= 12 :
    if i%3 != 0 :
        outString += str( i ) + ","  # convert to string, concatenate
    else :
        outString += "%d\n" % i      # add last value and terminate
        fp.write( outString )        #   with newline
        outString = ""
    i += 1

fp.close()
```

which produces the file `output.csv` with the following contents:

```
1,2,3
4,5,6
7,8,9
10,11,12
```

9 References

1. *Python Essential Reference, 3rd ed*, David M. Beazley, Sams Publishing, 2006.

2. Python Documentation, 2.6.2, *Python Programming Language – Official Website*, URL:http://docs.python.org/download.html, retrieved: June 4, 2009.