Basics of Python - 2

by Kaustubh Vaghmare

(IUCAA, Pune)

E-mail: kaustubh[at]iucaa[dot]ernet[dot]in

Our First Program - Rewritten!

Let us introduce the following modifications to the program.

- We use floats instead of ints.
- We accept the numbers from the user instead of "hard coding" them.

In [1]:
```
# Modified first program.
a = raw_input("Please enter number 1: ")
b = raw_input("Please enter number 2: ")

c, d = a+b, a-b
q, r = a/b, a*b

print c,d,q,r
```

```
Please enter number 1: 5.0
Please enter number 2: 2.5
```

```
---------------------------------------------------------------
------------
TypeError                                 Traceback (most recen
t call last)
<ipython-input-1-82ce9ef7d8e0> in <module>()
      3 b = raw_input("Please enter number 2: ")
      4
----> 5 c, d = a+b, a-b
      6 q, r = a/b, a*b
      7

TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

What happened?

- Anything input through the keyboard using raw_input() is ... a "string".
- Strings support addition (concatenation) but nothing else.

So what should we do?

- "3.0" is a string. 3.0 is a float!
- To convert "3.0" into a string, we use a simple function float("3.0")

So, let's rewrite our program!

In [2]:
```
a = float( raw_input("Enter Number 1: ") )
b = float( raw_input("Enter Number 2: ") )

c,d = a+b, a-b
q,r = a*b, a/b

print "Addition = %f, Difference = %f " % (c,d)
print "Division = %f, Quotient = %f" % (q,r)
```

```
Enter Number 1: 5.0
Enter Number 2: 2.5
Addition = 7.500000, Difference = 2.500000
Division = 12.500000, Quotient = 2.000000
```

Yuck! That ugly output! Wish I could control the decimal places...

In [3]:
```
a = float( raw_input("Enter Number 1: ") )
b = float( raw_input("Enter Number 2: ") )

c,d = a+b, a-b
q,r = a*b, a/b

print "Addition = %.2f, Difference = %.2f " % (c,d)
print "Division = %.2f, Quotient = %.2f" % (q,r)
```

```
Enter Number 1: 5.0
Enter Number 2: 2.5
Addition = 7.50, Difference = 2.50
Division = 12.50, Quotient = 2.00
```

Ah! Now, that's much better.

String Formatting

We have seen a powerful of constructing strings in the previous example.

In [4]:
```
print "Addition = %.2f, Difference = %.2f " % (c,d)
```

```
Addition = 7.50, Difference = 2.50
```

C / FORTRAN users will immediately understand this method of string construction.

Python supports this and its own way of string formatting.

In [5]:
```
gal_name = "NGC 7709"; int_bmagnitude = 13.6
```

In [6]:
```
statement1 = "The galaxy %s has an integrated \
B-band magnitude of %.2f" % (gal_name, int_bmagnitude)
```

In [7]:
```
statement2 = "The galaxy {0:s} has an integrated \
B-band magnitude of {1:.2f}".format(gal_name, int_bmagnitude)
```

In [8]:
```
statement3 = "The galaxy {name:s} has an integrated \
B-band magnitude of {mag:.2f}".format(name=gal_name, mag=int_bma
gnitude)
```

All the above statements are equivalent!

In [15]:
```
print statement1, "\n", statement2, "\n", statement3, "\n"
```

```
The galaxy NGC 7709 has an integrated B-band magnitude of 13.60

The galaxy NGC 7709 has an integrated B-band magnitude of 13.60

The galaxy NGC 7709 has an integrated B-band magnitude of 13.60
```

You can choose whichever method you like!

As a former C/C++ user, I'd prefer the first method.

But ... second and third methods are more "Pythonic".

Conditionals

In [16]:
```
num = int( raw_input("Enter number: ") )
if num %2 == 0:
    print "%d is even!" % num
else:
    print "%d is odd!" % num
```

```
Enter number: 3
3 is odd!
```

Let us write something bigger...

In [2]:
```
model_choice = int(raw_input( "Enter choice [1 or 2]: ") )
spectra = 3 # In realistic case, this will be some complicated o
bject.

if model_choice == 1:
    model1(spectra)
    print "Model 1 fitted."
elif model_choice == 2:
    model2(spectra)
    print "Model 2 fitted."
else:
    print "Invalid model entered."
```

```
Enter choice [1 or 2]: 1
Model 1 fitted.
```

What do you notice apart from the syntax in the above example?

Indentation - A Vital Part of the Pythonic Way

Be it the if-block illustrated above or the loops or the functions (to come soon), indentation is at the heart of the Python's way of doing things!

Function definitions, loops, if-blocks - nothing has your typical boundaries like { } as in C/C++/Java.

The "level of the indentation" defines the scope of a "block".

In support of indentation

Look at the following C-like code.

```
if (x>0)
    if (y>0)
        print "Woohoo!"
else
        print "Booboo!"
```

Which "if" does the "else" belong to?

In C like languages, the braces {}s do the marking, the indentation is purely optional. In Python, indentation levels determine scopes. In Python the "the else" belongs to "if (x>0)".

Python forces you to write clean code! (Obfuscation lovers, go to hell!)

Wrapping up if-elif-else

The general syntax:

```
if <condition>:
    do this
    and this
elif <condition>:
    this
    and this
...
else:
    do this
    and this
```

Conditions are anything that return True or False.

- == (equal to)
- !=
-
- >=
- <
- <=

You can combine conditionals using "logical operators"

- and
- or
- not

The Boolean Data Type

In [3]:
```
a = True
b = False

if a:
    print "This comes on screen."

if b:
    print "This won't come on screen."
```

```
This comes on screen.
```

In [4]:
```
type(a) # To check type of object.
```

Out[4]:
```
bool
```

Almost Everything has a Boolean Equivalent

In [5]:
```
a = 1
b = 0

if a:
    print "Hello!"
if b:
    print "Oh No!"
```

```
Hello!
```

In [8]:
```
s1 = ""; s2 = "Hello"

if s1:
    print "Won't be printed."
if s2:
    print "Will be printed."
```

```
Will be printed.
```

Conditional Expression

Consider...

In [9]:
```
if 5 > 6:
    x = 2
else:
    x = 3
```

In [10]:
```
y = 2 if 5 > 6 else 3
```

In [11]:
```
print x,y
```

```
3 3
```

A Second Tour of the Data Types

The two other data types we need to know:

- Lists
- Dictionaries

Data Types we will not cover (formally):

- Tuples (immutable lists!)
- Sets (key-less dictionaries!)
- Complex Numbers
- Fractions
- Decimals
- Ordered Tuples ...

Lists

In [12]:
```
a = [1,2,3,4] # simple ordered collection
```

In [13]:
```
b = ["Hello", 45, 7.64, True] # can be heterogeneous
```

In [14]:
```
a[0], a[-1], a[1:3] # All "sequence" operations supported.
```

Out[14]:
```
(1, 4, [2, 3])
```

In [15]:
```
b[0][1] # 2nd member of the 1st member
```

Out[15]:
```
'e'
```

In [16]:
```
a = [ [1,2,3] , [4,5,6] , [7,8,9] ] # list of lists allowed.
```

In [17]:
```
a[2][1] # Accessing elements in nested structures.
```

Out[17]: 8

In [18]:
```
[1,3,4] + [5,6,7] # Support concatenation
```

Out[18]: [1, 3, 4, 5, 6, 7]

In [19]:
```
[1,6,8] * 3 # Repetition (like strings)
```

Out[19]: [1, 6, 8, 1, 6, 8, 1, 6, 8]

Lists are Mutable! (Strings are not!)

In [20]:
```
a = [1,4,5,7]
```

In [21]:
```
print a
```

```
[1, 4, 5, 7]
```

In [22]:
```
a[2] = 777
```

In [23]:
```
print a
```

```
[1, 4, 777, 7]
```

List Methods

In [27]:
```
a = [1,3,5]
print a
```

```
[1, 3, 5]
```

In [28]:
```
a.append(7) # adds an element to the end
print a # the list has changed (unlike string methods!)
```

```
[1, 3, 5, 7]
```

In [29]:
```
a.extend([9,11,13]) # concatenates a list at the end
print a
```

```
[1, 3, 5, 7, 9, 11, 13]
```

In [30]:
```
print a
```

```
[1, 3, 5, 7, 9, 11, 13]
```

In [31]:
```
a.pop() # Removes one element at the end.
print a
```

```
[1, 3, 5, 7, 9, 11]
```

In [32]:
```
a.pop(2) # Removes 3rd element.
print a
```

```
[1, 3, 7, 9, 11]
```

Don't Forget!!!

In [33]:
```
print dir(a) # list of methods for a list "a"
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__deli
tem__', '__delslice__', '__doc__', '__eq__', '__format__', '__g
e__', '__getattribute__', '__getitem__', '__getslice__', '__gt_
_', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__',
 '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__
rmul__', '__setattr__', '__setitem__', '__setslice__', '__sizeo
f__', '__str__', '__subclasshook__', 'append', 'count', 'extend
', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

In [34]:
```
help(a.sort)
```

```
Help on built-in function sort:

sort(...)
    L.sort(cmp=None, key=None, reverse=False) -- stable sort *I
N PLACE*;
    cmp(x, y) -> -1, 0, 1
```

Implications of Mutability

In [35]:
```
l = [1,2,3,4]
m = l

l.append(5)
print l
print m
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

l and m point to the same object. When the object mutates, whether you refer to it using l or m, you get the same mutated object.

How do I make a copy then?

In [36]:
```
l = [1,2,3,4]
m = l[:]

l.append(5)
print l
print m
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
```

Python has a module called "copy" available for making copies. Will be covered later.

Dictionaries

- Imagine a list as a collection of objects obj0, obj1, obj2 ...
- First object has a location 0, second 1 ...
- Now, imagine renaming location 0 as "something", location 1 as "somethingelse" ...
- Earlier, you accessed objects at numbered locations a[0].
- Now, you access objects by specifying location names a["something"]

Let's see this at work.

In [37]:
```
d1 = { "a" : 3, "b" : 5}
print d1["a"]
print d1["b"]
```

```
3
5
```

"a", "b" are called keys and 3,5 are called values. So formally, a dictionary is a collection of key-value pairs.

In [38]:
```
d1["c"] = 7 # Since "c" does not exist, a new key-value pair is
made.
d1["a"] = 1 # SInce "a" exists already, value is modified.
print d1 # You will notice the order is not the same.
```

```
{'a': 1, 'c': 7, 'b': 5}
```

Dictionary Methods

In [39]:
```
keys = d1.keys() # Returns a list of all keys which is stored in
 "keys".
print keys
```

```
['a', 'c', 'b']
```

In [40]:
```
values = d1.values() # Returns a list of values.
print values
```

```
[1, 7, 5]
```

In [41]:
```
d1.items() # List of Tuples of key-value pairs.
```

Out[41]:
```
[('a', 1), ('c', 7), ('b', 5)]
```

Defining Dictionaries - ways to do this

In [42]:
```
d1 = {"a":3, "b":5, "c":7} # we've seen this.
```

In [43]:
```
keys =  ["a", "b", "c"]
values = [3,5,7]
d2 = dict( zip(keys,values) ) # creates dictionary similar to d2
```

In [44]:
```
d3 = dict( a=3, b=5, c=7) # again, same as d1,d2
```

In [45]:
```
d4 = dict( [ ("a",3), ("b",5), ("c",7) ] ) # same as d1,d2,d3
```

Loop Loop Loop

In [46]:
```
x = 0
while x<5:
    print x, # NOTICE the comma at the end. Suppresses new line.
    x += 1
```

```
0 1 2 3 4
```

In [49]:
```
x = 1
while True:
    print "x = %d" % x
    choice = raw_input("Do you want to continue? ")
    if choice != "y":
        break # This statement breaks the loop.
    else:
        x += 1
```

```
x = 1
Do you want to continue? y
x = 2
Do you want to continue? y
x = 3
Do you want to continue? q
```

The "for" loop - Pay Attention!

In [51]:
```
x = [5,6,7,8,9,0] # a simple list
for i in x:
    print i
```

```
5
6
7
8
9
0
```

In " for i in x", x can be anything that is a collection of things.

In [52]:
```
s = "Hello!"

for c in s:
    print c
```

```
H
e
l
l
o
!
```

No No No! I my good old for-loop back which generates numbers x to y in steps of z!!!

In [53]:
```
# OKAY!!!

for i in range(2,15,3):
    print i
```

```
2
5
8
11
14
```

In [54]:
```
range(10)
```

Out[54]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [55]:
```
range(2,10)
```

Out[55]:
```
[2, 3, 4, 5, 6, 7, 8, 9]
```

Let us see some wicked for-loops.

In [56]:
```
a = [1,2,3,4,5]
b = "Hello"
c = zip(a,b)
print c

for i,j in c:
    print i, j
```

```
[(1, 'H'), (2, 'e'), (3, 'l'), (4, 'l'), (5, 'o')]
1 H
2 e
3 l
4 l
5 o
```

In [57]:
```
a = "Hello!"

for i, c in enumerate(a):
    print "Character no. %d is %s" % (i+1, c)
```

```
Character no. 1 is H
Character no. 2 is e
Character no. 3 is l
Character no. 4 is l
Character no. 5 is o
Character no. 6 is !
```

You can break and continue for-loops too!

In [60]:
```
for i in range(10000):
    if i%2 == 0: # Even
        print "Even"
        continue
    print "Odd!"

    if i == 7: # What if I had said "i==8 or i==10" ??????
        break
```

```
Even
Odd!
Even
Odd!
Even
Odd!
Even
Odd!
```

Traversing Dictionaries using for-loops

In [61]:
```
d = dict( a = 1, b = 2, c = 3, d = 4)

for key,value in d.items():
    print key, "-->", value
```

```
a --> 1
c --> 3
b --> 2
d --> 4
```

In [63]:
```
for key in d.keys():
    print key, "-->", d[key]
```

```
a --> 1
c --> 3
b --> 2
d --> 4
```

Function Basics

In [64]:
```
def myfun():
    print "Hello World!"
```

In [65]:
```
myfun()
```

```
Hello World!
```

In [66]:
```
x = myfun()
print x
```

```
Hello World!
None
```

Functions with Arguments

In [67]:
```
def myfun(string):
    print string
```

In [68]:
```
myfun() # ERROR
```

```
---------------------------------------------------------------------
------------
TypeError                                 Traceback (most recen
t call last)
<ipython-input-68-f3ab186f5d61> in <module>()
----> 1 myfun() # ERROR

TypeError: myfun() takes exactly 1 argument (0 given)
```

In [69]:
```
myfun("Happiness!")
```

```
Happiness!
```

Function with a Return Value

In [70]:
```
def myfun(a,b):
    return a+b
```

In [71]:
```
x = myfun(2,3)
print x
```

```
5
```

Function with Optional Arguments

In [72]:
```
def myfun( string = "Hello World!"):
    print string
```

In [73]:
```
myfun() # No argument supplied.
```

```
Hello World!
```

In [74]:
```
myfun("Not in a Mood!")
```

```
Not in a Mood!
```

In [76]:
```
x = "I am a string!"
myfun(x)
```

```
I am a string!
```

Functions are Objects!

In [77]:
```
import math
print math.sqrt(5)

a = math.sqrt
print a(5)
```

```
2.2360679775
2.2360679775
```

In [78]:
```
def do(f,x):
    f(x)
do(myfun, "Hello!")
```

```
Hello!
```

Handling Files

Let us study how to handle files through a simple exercise. The basic approach involves creating file objects in Python and use various methods associated with file objects to handle file I/O.

- open() function is used to create file object.
- fileObject.read() - reads entire file as one big string.
- fileObject.write() - to write a string in a file.
- fileObject.readlines() - to read each line as an element of a list.
- fileObject.writelines() - to write a set of lines, each one being a string.
- fileObject.close() - to close a file (buffer flush)

Program to "Double Space" a File

In []:
```
"""
Program to create a double spaced file.
Input: File Name
Output: Modified File with .sp extension
"""

import sys # we need this to parse command line arguments.
import os # we need this to check for file's existence
```

In []:
```
# Check number of arguments.
if len(sys.argv) == 2:
        infile_name = sys.argv[1]
else:
        print "Oops! Incorrect Number of Arguments."
        sys.exit(2)

# Check if file exists.
if not os.path.isfile(infile_name):
        print "File doesn't exist."
        sys.exit(3)
```

In []:
```
# Open the input file.
infile = open(infile_name, "r")

# Open an output file.
outfile = open(infile_name + ".sp", "w")

# Loop over each line, add new line to each line.
for line in infile.readlines():
        line = line+"\n"
        outfile.write(line)

outfile.close()
infile.close()
```