Other Topics in Python

by

Kaustubh Vaghmare

(IUCAA, Pune)

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

Installing Packages / Modules

There are two ways of installing third party modules.

- Downloading source and installing as per instructions.
- Using "pip", a package manager for Python.

NOTE: Windows and Mac OS X users are on your own. I've no idea how to handle these two platforms.

For a complete list of registered Python packages : https://pypi.python.org/pypi

A Typical Package Installation from Source

- Download source.
- Unzip it.
- There is generally a setup.py. All you need to do is,

  python setup.py install

You may or may not have to change the **PYTHONPATH**.

Using ”pip”

"pip" is a program for installing any program registered in the Python package index.
First you need to install the program. For Debian based Linux distros,

```
sudo apt-get install python-pip
```

If successful, you can do the following,

```
pip install [packagename]
```

Example:

```
pip install matplotlib
```

List Comprehensions

These are special ways of creating new lists in a compact syntax.

In [25]:
```
squares = []
for i in range(10):
    squares.append(i**2)
print squares
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Now, consider the following equivalent syntax.

In [26]:
```
squares_alt = [ i**2 for i in range(10) ]
print squares_alt
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

With an if-condition

In [27]:
```
squares_even = [ i**2 for i in range(10) if i%2 == 0 ]
print squares_even
```

```
[0, 4, 16, 36, 64]
```

We can use this for filtering existing lists.

In [28]:
```
a = [ "Hello", "Glenn", "Man", "Lady", "Howard", "Leonard"]
a_len5 = [ i for i in a if len(i) == 5 ]
print a_len5
```

```
['Hello', 'Glenn']
```

Nested List Comprehensions!

In [29]:
```
k = [ (i,j) for i in range(3) for j in range(3) ]
print k
```

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1)
, (2, 2)]
```

The equivalent syntax in traditional loops would be -

In [30]:
```
k = []
for i in range(3):
    for j in range(3):
        k.append( (i,j) )
print k
```

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1)
, (2, 2)]
```

List Comprehensions - Dos and Donts

- For simple for-loops like we saw earlier, you SHOULD use list comprehensions.
- In fact, list comprehensions run faster than traditional loops.

But,

- List comprehensions, especially nested ones, can obfuscate code.
- In such a case, it's better to use elaborate loops.

Examples of List Comprehensions at Work

Consider we wanted to read a list of words from a file, where each word is on one line. You might want to say,

In [31]:
```
f = open("words.txt")
words = f.readlines()
print words
```

```
['First\n', 'Second\n', 'Third\n', 'Fourth\n', 'Fifth\n']
```

Notice the "" at the end of each string in the list. We would like to "not have them".

In [32]:
```
words = [ i.rstrip() for i in open("words.txt").readlines() ]
print words
```

```
['First', 'Second', 'Third', 'Fourth', 'Fifth']
```

Consider replacing all NaN values with zeros in a list.

In [33]:
```
data = [1, 2, 3, 4, "NaN", 7, 4, "NaN"]
data_new = [ i if i!="NaN" else 0 for i in data ]
```

In [34]:
```
print data_new
```

```
[1, 2, 3, 4, 0, 7, 4, 0]
```

Functions - Arguments - Keyword Arguments

In [35]:
```
def f(**kwargs):
    print kwargs

f(a=3, b=5, c=7)
```

```
{'a': 3, 'c': 7, 'b': 5}
```

This way, it is possible to make functions which accept arbitrary number of arguments but with names.

NOTE: Unlike *args, where one gets a list of elements, **kwargs, one gets a dictionary of arguments.

Scopes

Let us spend some time understanding variable scopes.

In [36]:
```
x = 3
def f():
    print x
```

In [37]:
```
f() # calling the function.
```

```
3
```

In [38]:
```
x = 3
def f():
    x = 2
    print x # This is the local x
print x # This is the global x
```

```
3
```

In [39]:
```
f()
```

```
2
```

Let us see a more complicated example.

In [40]:
```
x = 3
def f():
    x = 4
    def g():
        x = 5
        print x # this uses 'x' within the block.
    print x # the local "x"
    g()
print x # Global 'x'
```

```
3
```

In [41]:
```
f()
```

```
4
5
```

The "global" statement.

In [42]:
```
x = 44
def f():
    global x
    x = 33
    print x
print x
```

```
44
```

In [43]:
```
f()
```

```
33
```

In [44]:
```
print x
```

```
33
```

Choose names wisely!

In [45]:
```
a = "Some String"
print type(a)
```

```
Some String
None
```

In [46]:
```
def type(x):
    print x
type(a)
```

```
Some String
```

L E G B : Local, Enclosing, Global, Built-in

Therefore, if you define a new variable with same name as built-in, it will override it because of the above order of looking up variables.

Crazy Crazy World

Here is some crazy behavior.

In [47]:
```
x = 3
def f():
    print x # you expect this to be global.
    x = 4 # and now this x to be local.
```

In [48]:
```
f()
```

```
---------------------------------------------------------------
------------
UnboundLocalError                         Traceback (most recen
t call last)
<ipython-input-48-0ec059b9bfe1> in <module>()
----> 1 f()

<ipython-input-47-865880ab4ad6> in f()
      1 x = 3
      2 def f():
----> 3     print x # you expect this to be global.
      4     x = 4 # and now this x to be local.
      5

UnboundLocalError: local variable 'x' referenced before assignm
ent
```

Some Useful Built-in Functions

Map it!

In [49]:
```
def increment(x):
    return x+1

a = [1,2,3,4]
b = map(increment, a)
print b
```

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

So, map() applies a function to a list element-wise and returns the resulting list.

Zip it up.

In [50]:
```
a = [1,2,3]
b = [4,5,6]
c = zip(a,b)
print c
```

```
[(1, 4), (2, 5), (3, 6)]
```

In [51]:
```
x = [1,2,3]
y = [4,5,6,7] # inequal length of x and y
z = zip(x,y)
print z
```

```
[(1, 4), (2, 5), (3, 6)]
```

Filter it!

In [52]:
```
def gt(x):
    if x > 0:
        return True

a = range(-5,5)
a_new = filter(gt, a)
print a_new
```

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

Philosophy of Python

In [53]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to
do it.
Although that way may not be obvious at first unless you're Dut
ch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea
.
Namespaces are one honking great idea -- let's do more of those
!
```

The PEP-8 Style Guide

Can be accessed at:

http://legacy.python.org/dev/peps/pep-0008/

It gives out various conventions to follow such as how to name your variables, how to import modules and so on...

If you especially intend to share your code in open source projects, it will be better for you to follow as many guidelines as possible from this document. It helps make your programs "look good"!

Is it a Module? Program? Or Both

Consider the following situation. You have a program...

... the program should behave differently when used as a module ...

... and differently when executed directly ... possible?

YES!

Let's see how!

Let us revisit our module.

```
"""
This is a custom module.
Containing some functions for the purpose of demonstration.
"""
def fun1():
    print "Inside fun1"

def fun2():
    print "Inside fun2"

pi = 3.14
e = 2.7

print "I am a Custom Module"
```

Let us run our program directly.

In [54]:
```
%run Example.py
```

```
I am a Custom Module
```

And let us import it!

In [55]:
```
import Example
```

```
I am a Custom Module
```

No difference! It does not make sense for **import Example** to generate output, so let's see how we can fix that.

```
"""
This is a custom module.
Containing some functions for the purpose of demonstration.
"""
def fun1():
    print "Inside fun1"

def fun2():
    print "Inside fun2"

pi = 3.14
e = 2.7

if __name__ == "__main__": # Following will run only if program is execute
d directly!
    print "I am a Custom Module"
```

Now, **import Example** will not print *I am a Custom Module*. But, directly executing the script will!

Hash Bang!

It is possible to make Python scripts behave like shell scripts on the Linux platform. The first line of the program should contain the following:

```
#! /usr/bin/python
```

You can then name your file as **DoThis**. Then make the file executable.

```
$ chmod u+x DoThis
```

And to execute, you can directly say,

The ”in” keyword:

Useful to check the memberships in collections.

In [1]:
```
a = [1,2,3,4,5]
2 in a
```

Out[1]: True

In [2]:
```
"H" in "Hello"
```

Out[2]: True

In [3]:
```
1 in {"a": 1, "b":2 }
```

Out[3]: False

In [4]:
```
"a" in {"a": 1, "b":2 }
```

Out[4]: True

Sets

A set can be initialized like this.

In [5]:
```
a = {1,2,3,4,5,5,5}; b = {4,6,7,7,8,9}
print a, b
```

```
set([1, 2, 3, 4, 5]) set([8, 9, 4, 6, 7])
```

In [6]:
```
a.intersection(b)
```

Out[6]:
```
{4}
```

In [7]:
```
a - b
```

Out[7]:
```
{1, 2, 3, 5}
```

In [8]:
```
b - a
```

Out[8]:
```
{6, 7, 8, 9}
```

In [9]:
```
a.union(b)
```

Out[9]:
```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Here is a cool technique for removing repetitions in a list.

In [11]:
```
l = [1,2,3,4,5,6,6,7,8,8,8,8]
print l
```

```
[1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 8, 8]
```

In [13]:
```
l = list(set(l))
print l
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

A Numpy equivalent would be np.unique()

Complex Numbers

In [14]:
```
a = 1+2j
```

In [15]:
```
type(a)
```

Out[15]:
```
complex
```

In [16]:
```
a**2
```

Out[16]:
```
(-3+4j)
```

In [17]:
```
a.conjugate()
```

Out[17]:
```
(1-2j)
```

Fractions

In [22]:
```
import fractions
x = fractions.Fraction(2,3)
y = fractions.Fraction(2,5)
print x+y
```

```
16/15
```

In [25]:
```
print x*y
```

```
4/15
```

In [27]:
```
fractions.gcd(45,54)
```

Out[27]:
```
9
```

The Take Home Message

Python's standard libraries have an amazing collection of functions for doing all sorts of things.

You can find all the modules and their documentation at : https://docs.python.org/2/library/index.html

As you have seen, creating a library is so easy in Python that - even if there is something Python does not do, you can make a module to support such a functionality. Who knows - if you do a good job, Python Software Foundation might even make it a part of the standard library.