More Python - Functions and Modules

by

Kaustubh Vaghmare

(IUCAA, Pune)

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

Functions

Blocks of code that perform a specific task.

In Python, a function is defined using the "def" keyword.

We have already seen examples of functions.

- float(), dict(), list(), len() etc.
- math sqrt(), floor(), ceil(), radians(), sin()
- open(), type() etc.

A Simple Function

```
In [13]: def myfun():
    print "Hello World!"
    print "Nice to see you."
    print "Outside the function."
```

Outside the function.

Pay attention to how the statements indented one level up are part of the function while the statement indented at the same level is not a part of the function.

In [14]: myfun() # This is how you call our function.

Hello World! Nice to see you.

Function With One Argument

```
In [15]: def myfun(a):
    print "Inside MyFun!"
    print a
In [16]: myfun() # WILL GIVE ERROR.
TypeError
t call last)
    <ipython-input-16-94e8ef4e305f> in <module>()
    ----> 1 myfun() # WILL GIVE ERROR.
```

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

As per function definition, one argument / input is needed. An attempt to call the function with none gives an error. EVEN supplying two arguments is wrong.

In [17]: myfun("An Input")

Inside MyFun!
An Input

REMEMBER

Python is a dynamically typed language. The true strength of this lies in the fact that you can also call the above function with a float or integer or list input!



Functions that "return" something.

In [20]: def add(a,b): return a+b In [23]: a = add(2,3) print a 5

A function that does not have a return statement returns by default something called "None".

```
In [24]: b = myfun("Hello")
Inside MyFun!
Hello
In [25]: print b
None
```

Functions can return more than one value at a time!

In [26]: def sumprod(a,b):
 return a+b, a*b

In [27]: s, p = sumprod(2,3)

Well, technically - Python is returning only one object but that one object is a tuple - in the above case - (2,3)

Optional Arguments

"I want a function to assume some values for some arguments when I don't provide them!" Let's see how this is achieved.

In [28]:	<pre>def myfun(message = "Default Message"): print message</pre>
In [29]:	myfun("Hello World")
	Hello World

In [30]: myfun()

Default Message

Functions with Arbitrary Number of Arguments

In [31]:	<pre>def sumitall(*values): total = 0 for i in values: total += i return total</pre>
In [32]:	sumitall(2,3,4,5)
Out[32]:	14
In [33]:	<pre>sumitall(2,3,4)</pre>

Out[33]: 9

Mixture of Arguments

In [34]: sumitall() Out[34]: 0 In [35]: def sumitall2(val1, *values): total = val1 for i in values: total += val1 return total In [36]: sumitall2(2) Out[36]: 2 In [39]: sumitall2(2,3,4) Out[39]: 6

In [40]: sumitall2() # WILL GIVE AN ERROR.
TypeError
t call last)
<ipython-input-40-349bf319af91> in <module>()
----> 1 sumitall2()

TypeError: sumitall2() takes at least 1 argument (0 given)

This way, you can design functions the way you want by imposing both a minimum number of arguments and have flexibility of an arbitary number of them!

Functions are Objects

Like lists, dictionaries, ints, floats, strings etc you can pass functions to other functions since they are just objects.



In [7]: x = myfun # simple variable assignment
x("Hilo!")

Hilo!

Function Documentation

Recall using help(math.hypot) to get help on understanding how to use hypot() function. Can we design a function myfun() and ensure that help(myfun) also gives a nice "help" output?

```
In [41]: def myfun(a,b):
    """
    Input: Two Objects
    Output: Sum of the two input objects.
    """
    return a+b
```

In [42]: help(myfun)

Help on function myfun in module __main__:

```
myfun(a, b)
    Input: Two Objects
    Output: Sum of the two input objects.
```

When designing functions of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document what the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function of your own, it is always good to document when the function own, it is always good to document when the function own when

Modules

Modules can be considered as "namespaces" which have a collection of objects which which you can use when needed. For example, math modules has 42 objects including two numbers "e" and "pi" and 40 functions.

Every program you execute directly is treated as a module with a special name __main__.

So, all the variables you define, the functions you create are said to live in the namespace of __main__.

When you say the following, you are making the namespace of **math** available to you.

import math

To then access something inside **math**, you say

math.object

So what happens when you "import"

- Python interpreter searches for math.py in the current directory or the installation directory (in that order) and compiles math.py, if not already compiled.
- Next, it creates a handle of the same name i.e. "math" which can be used to access the objects living inside math.

In [47]: import math type(math)

Out[47]: module

Other way to "import"

In the above example, you are accessing objects inside **math** through the module object that Python created. It is also possible to make these objects become a part of the current namespace.

from math import *

In [48]: from math import *
radians(45) # no math.radians required.

```
Out[48]: 0.7853981633974483
```

WARNING: The above method is extremely dangerous! If your program and the module have common objects, the above statement with cause a lot of mix-up!

A Middle Ground

If there is an object you specifically use frequently and would like to make it a part of your main namespace, then,

from ModuleName import Object

In [1]: from math import sin
 print sin(1.54)

0.999525830605

NOTE: If you import the same module again in the same program, Python does not reload. Use reload(ModuleName) for reloading.

Aliases for a Module

If you have decided to access a module's objects from its own namespace, you can choose to alias the module with a name.

import numpy as np
np.array(...)

Another example,

```
import matplotlib.pyplot as plt
plt.plot(x,y)
```

The Python Module Ecosystem

There are three types of modules you will encounter in Python.

- Built-in Modules (come with any standard installation of Python)
- Third Party Modules (need to be installed separately)
- Your Own Modules (we'll see how to make them soon)

Built-in Modules

- sys contains tools for system arguments, OS information etc.
- os for handling files, directories, executing external programs
- re for parsing regular expressions
- datetime for date and time conversions etc.
- pickle for object serialization
- csv for reading CSV tables

and many many more ...

Third Party Modules

These need to be installed separately.

- numpy / scipy numerical plus scientific computing extensions to Python
- matplotlib using Python for plots
- mayavi for animations in 3D
- pandas for tabular data analysis
- astropy Python for Astronomers
- scikit-learn machine learning and classification tools for Python

Making your Own Modules

Very simple. Open a file, say, "MyModule.py"

Write code in the file.

If the file is in the present folder or on the PYTHONPATH, the following will work.

import MyModule
MyModule.something ...

- _NOTE 1: _ File name must have extension.py
- _NOTE 2: _ When importing extension must be dropped.

Example Module - Example.py

```
"""
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"
```

The above code is stored in Example.py. Let's see how to use it.

In [1]: import Example

I am a Custom Module

Notice the message printed by Example.py. This is to illustrate that any output generated by Example.py will appear on the screen.

In [2]:	<pre>print Example.pi</pre>
	3.14
In [3]:	<pre>Example.fun1()</pre>
	Inside funl

In [4]: help(Example)

Help on module Example:

NAME

Example

FILE

/home/kaustubh/Dropbox/IIST2014/Example.py

DESCRIPTION

This is a custom module. Containing some functions for the purpose of demonstration.

FUNCTIONS

fun1()

fun2()

DATA

e = 2.7 pi = 3.14 Notice the description. It is what you enclosed in the "docstring" at the beginning of the /AdvPython.slides.html?print-pdf module.

In []:	
--------	--