Python 101 (part 7): Dinner With A Hungry Giant

By Vikram Vaswani

This article copyright Melonfire 2000−2002. All rights reserved.

Table of Contents

Big Iron

Last time out, I introduced the fundamental unit of software abstraction − the function − and demonstrated it in the context of a Python program. I illustrated how functions can be used to package code into reusable modules which, when combined with arguments and return values, add a whole new level of flexibility to your code. Finally, I talked at some length about variable scope inside and outside functions, and discussed some of the tools Python offers to assist in optimal usage of functions.

However, packaging your code into functions is just the beginning. Python also allows you to create collections of shared code fragments, or "modules", which may be imported and used by any Python program. Powerful, flexible and very interesting, modules can be one of the most potent weapons in a Python developer's arsenal...so long as they're used correctly.

Over the course of this article, I'll be exploring Python modules in detail, together with examples of how to create, import and use modules in your own development activities. As if that wasn't enough, I will also be looking at some of the built−in modules that ship with Python, and demonstrating how they can speed up code development. So let's get going, shall we?

Mercury Rising

Like Alice in Wonderland, I'll start at the beginning − what's a module anyway?

Modules are a way to group related pieces of code together. They allow developers to create a logical container for variables and functions, such that these variables and functions can be used by other programs that require them.

The goal? Very simple: by making it possible to share code in this manner, Python immediately makes it easier to create reusable software, cutting down development and testing time. Take it one step further: by allowing developers to create modules and providing the underpinnings to import them into other programs, Python ensures that a single copy of a module is in use across a system. This simplifies code maintenance by restricting updates and upgrades to a single file.

In the content of Python programming, a module is essentially a text file, ending in a .py extension and containing executable program code. The name of the file is treated as the name of the module; once the module has been imported into a Python program, this name is used in all subsequent references to the module.

Let's illustrate how this works by creating a simple module. Pop open your favourite text editor, and create a new text file containing the following function:

```
# define a function
def tempConv(temperature, scale):
if (scale == "c"):
fahrenheit = (temperature * 1.8) + 32
return fahrenheit
elif (scale == "f"):
celsius = (temperature − 32) / 1.8
return celsius
else:
return "Cannot convert!"
```

Save this file as "temperature.py"

Notice that the module does not require the name of the interpreter to be specified as the first line of the script, as do regular Python scripts.

You can now pull this module into any Python program − I'll illustrate how with the command−line interpreter.

```
Python 1.5.2 (#1, Aug 25 2000, 09:33:37) [GCC 2.96 20000731
(experimental)] on linux−i386
Copyright 1991−1995 Stichting Mathematisch Centrum, Amsterdam
>>> import temperature
>>> temperature.tempConv(98.6, "f")
37.0
>>> temperature.tempConv(37, "c")
98.6
>>> temperature.tempConv(37, "d")
'Cannot convert!'
>>>
```

Between A Rock And...Another Rock

The elements within a module − variables, functions, other imports − are referred to as "module attributes".

The "import" keyword is used to search for the named module and load it if found. Once the module has been successfully imported, functions and variables contained within it can be accessed by prefixing their names with the module name. Since each module sets up an independent namespace, this is a handy way to avoid name clashes.

To see how this works, consider the following example, which uses two modules, each containing a function named rock():

```
def rock(): # module huey.py
print "Between a rock and a hard place"
```

```
def rock(): # module dewey.py
print "Rock on!"
```

Let's see what happens when I try to use these in a program:

```
>>> import huey
>>> import dewey
>>> huey.rock()
Between a rock and a hard place
>>> dewey.rock()
Rock on!
>>>
```

Since each module attribute has a unique prefix, it's possible to distinguish between attributes with the same name in different modules, and thereby avoid name conflicts.

You'll be interested to know that everything you do in Python takes place within the context of a module. Even commands typed into the interactive command−line interpreter take place within the bubble of Python's top−level module, named "__main__".

Love Bytes

The "import" statement looks for module files in the directories specified in the $PYTHONPATH environment variable. If the named module isn't found in these directories, it returns an error − as the following example demonstrates:

```
>>> import SQLfunctions
Traceback (innermost last):
File "", line 1, in ?
ImportError: No module named SQLfunctions
>>>
```

Once you add the directory containing the module to the $PYTHONPATH variable and restart the interpreter, Python should find and import your module without any errors (you can also modify the search path via the built−in "sys" module − more on this later.)

In case Python finds more than one module with the specified name, the first one is used.

The first time Python imports a module, it automatically compiles the module as saves it as bytecode; this bytecode file has the same name as the module file, but ends in a .pyc extension. These .pyc files are automatically recompiled if the module changes in any way. If you take a look at your working directory after importing and using one or more modules in a Python program, you'll notice a bunch of these .pyc files scattered around − Python stores these to speed up the module the next time it's called.

```
$ ls −l
−rw−rw−r−− 77 Jul 23 23:04 dewey.py
−rw−rw−r−− 181 Jul 23 23:05 dewey.pyc
−rw−rw−r−− 98 Jul 23 23:04 huey.py
−rw−rw−r−− 202 Jul 23 23:22 huey.pyc
−rw−rw−r−− 244 Jul 23 22:49 temperature.py
−rw−rw−r−− 435 Jul 23 22:50 temperature.pyc
−rwxrwxr−x 222 Jun 8 10:50 module_test.py
```

If you play with this a little, you'll see that I told a little white lie a few paragraphs back, when I said that

Python could only import a module if it was located in a directory named in $PYTHONPATH. In fact, even if the module isn't located in the search path, but its corresponding bytecode is present, Python will still import and use the module.

Consider the following example, where the working directory (which is in the search path) doesn't contain the module source, but *does* contain the compiled bytecode version − Python goes ahead and uses the bytecode version without blinking:

```
$ ls -l
-rw-rw-r-- 435 Jul 23 22:50 temperature.pyc
-rwxrwxr-x 222 Jun 8 10:50 module_test.py



$ python
>>> import temperature
>>> temperature.tempConv(95, "f")
35.0
>>>
```

Enter The Hungry Giant

The first time a module is imported, the code within it is automatically executed. This comes in handy if you need to initialize variables, or print a copyright notice:

```
# menu.py



# set up dictionaries
breakfast = {'Mon':'Ham and Eggs', 'Tue':'Grilled Sandwiches',
'Wed':'Spanish Omelettes', 'Thu':'Bacon and Eggs',
'Fri':'Pancakes',
'Sat':'Scrambled Eggs', 'Sun':'Coffee and Donuts'}



lunch = {'Mon':'Russian Salad', 'Tue':'Fish and Chips',
'Wed':'Chicken
Curry', 'Thu':'Egg Salad', 'Fri':'Cheeseburgers',
'Sat':'Steak',
'Sun':'Stir-fried Chicken'}



dinner = {'Mon':'Pasta', 'Tue':'Thai Noodles', 'Wed':'Pork
Chops',
'Thu':'Prawns in Butter Garlic Sauce', 'Fri':'Fried Fish',
'Sat':'Mongolian
Chicken', 'Sun':'Vegetable Stew'}



# functions to return menu items based on day
def getBreakfastItem(day):
print "Breakfast on " + day + " is: " + breakfast[day]



def getLunchItem(day):
print "Lunch on " + day + " is: " + lunch[day]
```

```
def getDinnerItem(day):
print "Dinner on " + day + " is: " + dinner[day]



def generateMenu(day):
print "Breakfast on " + day + " is: " + breakfast[day]
print "Lunch on " + day + " is: " + lunch[day]
print "Dinner on " + day + " is: " + dinner[day]



print "This module is owned by The Hungry Giant. Cook smart.
Eat healthy.
Die anyway."
```

```
>>> import menu
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
>>> import menu
>>> import menu
>>> import menu
>>>
```

Note that the code within the module is only executed the first time; subsequent attempts to import the module do not execute the code within it.

In case you need to re−run the module code, Python offers the reload() function, which reloads a module and executes the code within it again. When a module is reloaded, all module attributes are refreshed with their original values.

In order to illustrate this, let's import the "menu.py" module above and access one of its attributes.

```
>>> import menu
```

```
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
>>> menu.breakfast["Fri"]
'Pancakes'
>>>
```

Next, let's alter this attribute.

```
>>> menu.breakfast["Fri"] = "Jam and Toast"
>>> menu.breakfast["Fri"]
'Jam and Toast'
>>>
```

Note how re−importing the module has no effect whatsoever on the changed attribute,

```
>>> import menu
>>> menu.breakfast["Fri"]
'Jam and Toast'
>>>
```

while reloading it resets all attributes back to their initial values.

```
>>> reload(menu)
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
```

```
>>> menu.breakfast["Fri"]
'Pancakes'
>>>
```

The reload() function only works if the module has been successfully imported prior to calling it. An attempt to reload() a module which has not been previously imported will result in an error.

```
>>> # module not yet imported
>>> reload(menu)
Traceback (innermost last):
File "", line 1, in ?
NameError: menu
>>> # import it...
>>> import menu
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
>>> # and now try reloading it!
>>> reload(menu)
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.

>>>
```

The reload() function comes in handy if a module changes after it has been imported; it provides a quick and easy way to update the namespace during program execution.

From Python, With Love

Thus far, you've been importing modules as is, and using module attributes by referencing them with the module name as prefix. Python also offers an alternative method to selectively access and use module attributes − the "from" statement.

The "from" statement allows you to import specific attributes from a module into the current namespace. Since these attributes become part of the current namespace, it no longer becomes necessary to prefix them with the module name in order to use them.

Consider the following example, which demonstrates how this works.

```
>>> from menu import lunch
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
>>> lunch["Tue"]
'Fish and Chips'
>>>
```

In this case, the module variable "menu.lunch" is imported into the current namespace as the variable "lunch".

It's important to exercise caution when using the "from" statement − since "from" imports module attributes directly into the current namespace, you run the risk of overwriting current names when you use it. To illustrate this, consider the following simple Python program:

```
#!/usr/bin/python



def getDinnerItem(day):
print "Sorry, diner closed on " + day



getDinnerItem("Mon")
```

When you run this, the output reads

```
Sorry, diner closed on Mon
```

Now, look what happens when you import some names from the "menu.py" module
into this program:

```
#!/usr/bin/python

def getDinnerItem(day):
print "Sorry, diner closed on " + day

# imports
from menu import dinner
from menu import getDinnerItem

getDinnerItem("Mon")
```

When you run this program. the imported names will overwrite the names already existing in the namespace, resulting in the following output:

```
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
Dinner on Mon is: Pasta
```

Another important gotcha with "from": importing names using "from" implies
that changes to those names (after they have been imported) are not
reflected in the parent module. Consider the following module:

```
# numbers.py
x = 1
y = 2
z = x + y
```

Look what happens when I import these values into a script:

```
#!/usr/bin/python

# import
from numbers import x,y,z

# at this stage, z = x + y => z = 3
print z

# alter the value of imported x
x = 10
```

```
# z is still referring to the value of x in the module, so z
still = 3
print z



# the only way to get z to recognize the new value of x is to
redefine z in
context
z = x + y



# now z = 12
print z
```

Doing The Math

With these caveats in mind, the "from" statement provides a convenient way to import specific bits of a module into another program. Most of the time, it's used in connection with modules containing a large number of different functions; it's easier − and more optimal − to simply import the functions you need, rather than the entire module. Here's an example, using the built−in "math" module:

```
>>> from math import sqrt, exp
>>> sqrt(256)
16.0
>>> exp(0)
1.0
>>>
```

If you have good reason to do so, or simply like to experiment, you can use "from" to import everything from a module into the current namespace − here's how:

```
>>> from menu import *
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
>>> dinner
{'Fri': 'Fried Fish', 'Tue': 'Thai Noodles', 'Thu': 'Prawns in
Butter
Garlic Sauce', 'Sun': 'Vegetable Stew', 'Wed': 'Pork Chops',
'Mon':
'Pasta', 'Sat': 'Mongolian Chicken'}
>>> lunch
{'Fri': 'Cheeseburgers', 'Tue': 'Fish and Chips', 'Thu': 'Egg
Salad',
'Sun': 'Stir−fried Chicken', 'Wed': 'Chicken Curry', 'Mon':
'Russian
Salad', 'Sat': 'Steak'}
>>> breakfast
{'Fri': 'Pancakes', 'Tue': 'Grilled Sandwiches', 'Thu': 'Bacon
and Eggs',
'Sun': 'Coffee and Donuts', 'Wed': 'Spanish Omelettes', 'Mon':
```

```
'Ham and
Eggs', 'Sat':'Scrambled Eggs'}
>>> getLunchItem("Wed")
>>> 'Chicken Curry'
>>>
```

If you need to prevent certain module attributes from being imported with a "from module import *" statement, you can prefix the attribute name within the module with an underscore. This is a primitive technique, but it does work − as the following example demonstrates:

```
# menu.py

# set up dictionaries
# snip

_dinner = {'Mon':'Pasta', 'Tue':'Thai Noodles', 'Wed':'Pork
Chops',
'Thu':'Prawns in Butter Garlic Sauce', 'Fri':'Fried Fish',
'Sat':'Mongolian
Chicken', 'Sun':'Vegetable Stew'}

# functions to return menu items based on day
# snip
```

Now, I will be unable to access the "_dinner" attribute when I use "from" to import everything,

```
>>> from menu import *
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
```

```
anyway.
>>> lunch["Tue"]
'Fish and Chips'
>>> _dinner["Tue"]
Traceback (innermost last):
File "", line 1, in ?
NameError: _dinner
>>> dinner["Tue"]
Traceback (innermost last):
File "", line 1, in ?
NameError: dinner
>>>
```

although I will still be able to access it when I perform an "import" operation.

```
>>> import menu
This module is owned by The Hungry Giant. Cook smart. Eat
healthy. Die
anyway.
>>> menu._dinner
{'Fri': 'Fried Fish', 'Tue': 'Thai Noodles', 'Thu': 'Prawns in
Butter
Garlic Sauce', 'Sun': 'Vegetable Stew', 'Wed': 'Pork Chops',
'Mon':
'Pasta', 'Sat': 'Mongolian Chicken'}
>>>
```

It's also possible for modules to import each other − here's a "circle" module which uses functions imported from the "math" module.

```
# circle.py

def area(r):
```

```
from math import pi
area = pi * r * r
return area
```

```
>>> import circle
>>> circle.area(5)
78.5398163397
>>>
```

String Theory (And Other Interesting Stuff)

Python comes with a whole bunch of built−in modules, which can substantially reduce the time you spend on development. Here's a list of the most common and useful ones (some of these are only available in Python 2.x):

The "string" module handles common string operations,

```
>>> import string
>>> string.lower("HELLO")
'hello'
>>> string.center("HELLO", 80)
' HELLO
'
>>> string.split("The red wolf ate the green pumpkin", " ")
['The', 'red', 'wolf', 'ate', 'the', 'green', 'pumpkin']
>>>
```

while the "re" module matches regular expression via its match() and search() functions,

```
>>> import re
>>> re.search("at", "Batman − Dark Knight")

>>> re.findall("oo", "boom boom bang")
['oo', 'oo']
>>>
```

and the "difflib" and "filecmp" modules help in comparing strings, files and directories.

The "math" module does for numbers what the "string" module does for strings.

```
>>> import math
>>> math.sin(60)
-0.304810621102
>>> math.sin(30)
-0.988031624093
>>> math.sin(0)
0.0
>>> math.cos(0)
1.0
>>> math.tan(45)
1.61977519054
>>> math.cos(90)
-0.448073616129
>>> math.hypot(3,4)
5.0
>>> math.pow(2, 4)
16.0
>>> math.exp(0)
1.0
>>>
```

The "cmath" and "random" modules handle complex numbers and random numbers.

```
>>> import cmath
>>> import rand
>>> cmath.pi
3.14159265359
>>> cmath.e
2.71828182846
>>> rand.randrange(25,100)
54
>>> rand.rand()
12992
>>> rand.choice(["Rachel", "Monica", "Chandler", "Joey",
"Phoebe", "Ross"])
'Rachel'
>>> rand.choice(["Rachel", "Monica", "Chandler", "Joey",
"Phoebe", "Ross"])
'Monica'
>>> rand.choice(["Rachel", "Monica", "Chandler", "Joey",
"Phoebe", "Ross"])
```

```
'Joey'
>>>
```

The "calendar" module offers a bunch of functions to handle date−related tasks,

```
>>> import calendar
>>> calendar.isleap(2001)
0
>>> calendar.isleap(2000)
1
>>> calendar.weekday(2001,01,05)
4
>>> calendar.prcal(2001)
2001



January February March
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
1 2 3 4 5 6 7 1 2 3 4 1 2 3 4
8 9 10 11 12 13 14 5 6 7 8 9 10 11 5 6 7 8 9 10 11
15 16 17 18 19 20 21 12 13 14 15 16 17 18 12 13 14 15 16 17 18
22 23 24 25 26 27 28 19 20 21 22 23 24 25 19 20 21 22 23 24 25
29 30 31 26 27 28 26 27 28 29 30 31



April May June
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
1 1 2 3 4 5 6 1 2 3
2 3 4 5 6 7 8 7 8 9 10 11 12 13 4 5 6 7 8 9 10
9 10 11 12 13 14 15 14 15 16 17 18 19 20 11 12 13 14 15 16 17
16 17 18 19 20 21 22 21 22 23 24 25 26 27 18 19 20 21 22 23 24
23 24 25 26 27 28 29 28 29 30 31 25 26 27 28 29 30
30



July August September
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
1 1 2 3 4 5 1 2
2 3 4 5 6 7 8 6 7 8 9 10 11 12 3 4 5 6 7 8 9
```

```
9 10 11 12 13 14 15 13 14 15 16 17 18 19 10 11 12 13 14 15 16
16 17 18 19 20 21 22 20 21 22 23 24 25 26 17 18 19 20 21 22 23
23 24 25 26 27 28 29 27 28 29 30 31 24 25 26 27 28 29 30
30 31



October November December
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
1 2 3 4 5 6 7 1 2 3 4 1 2
8 9 10 11 12 13 14 5 6 7 8 9 10 11 3 4 5 6 7 8 9
15 16 17 18 19 20 21 12 13 14 15 16 17 18 10 11 12 13 14 15 16
22 23 24 25 26 27 28 19 20 21 22 23 24 25 17 18 19 20 21 22 23
29 30 31 26 27 28 29 30 24 25 26 27 28 29 30
31
>>>
```

while the "time" module handles time−related operations and conversions.

```
>>>import time
>>> time.time()
996052081.879
>>> time.localtime(time.time())
(2001, 7, 25, 14, 38, 19, 2, 206, 0)
>>>
>>> time.strftime("%A %d %B %Y", time.localtime(time.time()))
'Wednesday 25 July 2001'
>>>
```

The "fileinput" and "xreadlines" modules offer functions to read and process files efficiently, while the "os" module provides OS−dependent functions.

```
>>> import os
>>> os.getcwd()
'/home/vikram'
```

```
>>> os.getuid()
519
>>> os.getgid()
100
>>> os.uname()
('Linux', 'medusa.melonfire.com', '2.2.14−5.0', '#1 Tue Mar 7
21:07:39 EST
2000', 'i686')
>>>
```

The very powerful "os.path" module offers functions to manipulate and test pathnames.

```
>>> os.path.abspath('../')
'/home'
>>>
>>> os.path.exists('/tmp/unicorn')
0
>>> os.path.basename('/usr/local/apache')
'apache'
>>>
>>> os.path.isabs('/usr/local/apache')
1
>>> os.path.isabs('../')
0
>>>
```

The "pwd", "grp" and "crypt" modules offer access to the UNIX password and group files, and come in handy for user and group manipulation tasks.

```
>>> import pwd, grp
>>> pwd.getpwnam('vikram')
('vikram', 'x', 519, 100, '', '/home/vikram', '/bin/bash')
>>> pwd.getpwuid(519)
('vikram', 'x', 519, 100, '', '/home/vikram', '/bin/bash')
>>> grp.getgrall()
```

```
[('root', 'x', 0, []), ('bin', 'x', 1, ['bin', 'daemon']),
('daemon', 'x',
2, ['bin', 'daemon']), ('sys', 'x', 3, ['bin', 'adm']),
('adm', 'x', 4,
['adm', 'daemon']), ('tty', 'x', 5, []), ('disk', 'x', 6, []),
('lp', 'x',
7, ['daemon', 'lp']), ('mem', 'x', 8, []), ('kmem', 'x', 9,
[]), ('wheel',
'x', 10, [])]
>>>
```

The "cgi", "urllib" and "httplib" modules are used to connect your Python program to the Web; the "smtplib" and "poplib" modules help in writing mail clients; the "mimetools" module helps to process MIME−encapsulated email messages; and the "xmllib", "xml.dom" and "xml.sax" modules provide the architecture necessary to handle XML data. Whew!

In case you need to find out more about a specific module − or any other Python object − consider using the dir() function, which returns a list of object attributes.

```
>>> import string
>>> dir(string)
['__builtins__', '__doc__', '__file__', '__name__', '_idmap',
'_idmapL',
'_lower', '_re', '_safe_env', '_swapcase', '_upper', 'atof',
'atof_error',
'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
'capwords',
'center', 'count', 'digits', 'expandtabs', 'find',
'hexdigits', 'index',
'index_error', 'join', 'joinfields', 'letters', 'ljust',
'lower',
'lowercase', 'lstrip', 'maketrans', 'octdigits', 'replace',
'rfind',
'rindex', 'rjust', 'rstrip', 'split', 'splitfields', 'strip',
'swapcase',
'translate', 'upper', 'uppercase', 'whitespace', 'zfill']
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2',
'ceil', 'cos', 'cosh', 'e', 'exp', 'fabs', 'floor', 'fmod',
'frexp',
'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'sin',
'sinh',
'sqrt', 'tan', 'tanh']
>>> import cgi
>>> dir(cgi)
['FieldStorage', 'FormContent', 'FormContentDict',
'InterpFormContentDict',
'MiniFieldStorage', 'StringIO', 'SvFormContentDict',
'__builtins__',
'__doc__', '__file__', '__name__', '__version__', 'dolog',
'escape',
'initlog', 'log', 'logfile', 'logfp', 'maxlen', 'mimetools',
'nolog', 'os',
'parse', 'parse_header', 'parse_multipart', 'parse_qs',
'print_arguments',
'print_directory', 'print_environ', 'print_environ_usage',
'print_exception', 'print_form', 'rfc822', 'string', 'sys',
'test',
'urllib']
>>>
```

Every Python module has a name, which is exposed as the module attribute "__name__".

```
>>> import string
>>> string.__name__
'string'
>>> import math
>>> math.__name__
'math'
>>>
```

Remember when I told you that the default module for all your Python activities is "__main__"? You can use "__name__" to keep me honest...

```
>>> __name__
```

```
'__main__'
>>>
```

Bucking The System

A special mention should be made here of Python's "sys" module, which allows you to manipulate system−specific parameters like the list of currently−loaded modules and the module search path. The following example demonstrates the important attributes of this module:

```
>>> import sys
>>> # path to python binary
>>> sys.executable
'/usr/bin/python'
>>> # platform
>>> sys.platform
'linux−i386'
>>> # list of loaded modules
>>> sys.modules
{'os.path': ,
'os':
, 'readline': 'readline' from
'/usr/lib/python1.5/lib−dynload/readline.so'>,
'exceptions': '/usr/lib/python1.5/exceptions.pyc'>,
'__main__': (built−in)>, 'posix': , 'sys': (built−in)>,
'__builtin__': , 'site':
, 'signal': 'signal' (built−in)>, 'UserDict':
'/usr/lib/python1.5/UserDict.pyc'>, 'posixpath':
, 'stat':
}
>>>
```

The "path" attribute specifies the search path for modules, and can be modified using standard list constructs.

```
>>> sys.path
['', '/usr/lib/python1.5/',
'/usr/lib/python1.5/plat−linux−i386',
'/usr/lib/python1.5/lib−tk', '/usr/lib/python1.5/lib−dynload']
>>> sys.path.append("/usr/local/pymod/")
>>> sys.path
['', '/usr/lib/python1.5/',
'/usr/lib/python1.5/plat-linux-i386',
'/usr/lib/python1.5/lib-tk', '/usr/lib/python1.5/lib-dynload',
'/usr/local/pymod/']
>>>
```

The "argv" attribute contains arguments passed to the program on the command line. Consider the following program:

```
#!/usr/bin/python

import sys

print "Arguments:"

for x in sys.argv:
print x
```

Here's the output:

```
$ script.py medusa.server.com vikram 110
Arguments:
script.py
medusa.server.com
vikram
110
```

As you can see, the first element of the "argv" list contains the name of the called script, with the remaining elements holding the command−line arguments.

Finally, the "__builtin__" module contains information on the various functions that are built into the interpreter. Take a look:

```
>>> import __builtin__
>>> dir()
['__builtin__', '__builtins__', '__doc__', '__name__']
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'EOFError',
'Ellipsis',
'EnvironmentError', 'Exception', 'FloatingPointError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplementedError',
'OSError', 'OverflowError', 'RuntimeError', 'StandardError', 'SyntaxError',
'SystemError', 'SystemExit', 'TypeError',
'ValueError', 'ZeroDivisionError', '_', '__debug__', '__doc__',
'__import__', '__name__', 'abs', 'apply', 'buffer', 'callable', 'chr',
'cmp', 'coerce', 'compile', 'complex', 'delattr', 'dir', 'divmod', 'eval',
'execfile', 'exit', 'filter',
'float', 'getattr', 'globals', 'hasattr', 'hash', 'hex', 'id', 'input',
'int', 'intern', 'isinstance', 'issubclass', 'len', 'list', 'locals',
'long', 'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'quit', 'range',
'raw_input', 'reduce',
'reload', 'repr', 'round', 'setattr', 'slice', 'str', 'tuple', 'type',
'vars', 'xrange']
>>>
```

The errors you see are built−in exceptions − we'll be discussing them in detail in the next article.

More information on the various modules which ship with Python is available at http://www.python.org/doc/current/lib/lib.html

And that's about it for this discussion of Python modules. In this article, you found out how to logically group functions together into modules, which are the highest−level abstraction in Python. You examined two different techniques for importing module functions and variables into your own programs, together with the implications of each on your program's namespace. Finally, you found out a little bit more about the default modules that ship with Python, hopefully saving yourself some time the next time you sit down to code a Python program

In the next − and final − article of this series, I will be examining Python's error−handling routines, and demonstrating how to use them to trap and resolve program errors. Make sure that you come back for that one!

Note: All examples in this article have been tested on Linux/i586 with Python 1.5.2. Examples are illustrative only, and are not meant for a production environment. YMMV!