More Python Programming: Modules

Written by Dave Pawlowski, October 16, 2011

1 Introduction

By now, you've learned how to create functions, so that you can break up your code into segments that make testing and dubugging easier. Also, you've learned how to use Python's built in modules, so that you can get access to all sorts of useful functions that are already written. But what if you create a function that would be useful in more than one program? By creating and storing that function in a module, you can reuse code in multiple programs without having to copy the code into every file that you need it in.

2 Creating modules

Creating modules is easy. All that is necessary is to put your function definitions into a file that has the .py extension. In your cwd, create a file called tools.py:

```
from timing import *
import glob
import os
import pdb
import sys

expand = os.path.expanduser

def file_search(filename):
   """Find a file given a filename.  Unix style wildcards are acceptable.
   """
   filelist = glob.glob(expand(filename))
   return filelist
```

Save the file and start the python shell. Now, you can import this module like you would any other module.

```
>>> import tools
```

Now, you have access to the function `file_search`. But that's not all that we've done in tools.py. We've also imported a few other modules, glob, os, pdb, and sys, as well as renamed one of the functions with in the os.path module, expanduser.

Used in this way, modules are incredibly useful. If there are python packages that you use almost every time that you write a program, you can load them in your own module, and then just load your module. In this case, if I were writing a program, in order to get access to glob, os, pdb, and sys modules, all I would need to do is load tools. That's one line of code rather than 4.

Note that you can also create variables that you may want to use. In this case, you created expand, which is a function object. Variables defined inside a module are called attributes of the module, and the dot operator is used to access them, just like functions:

```
>>> print tools.expand
```

3 Docstrings

Another Python feature that we haven't talked about are documentation strings (or docstrings). Docstrings provide a really easy way of associating documentation with modules, functions, classes, and methods. An object's docsting is defined by including a string constant as the first statement in the object's definition. In the file_search funtion, the docstring is the text in between the triple quotes """.

Python treats this code in a special manner. If there is a docstring, python defines an attirbute of the object that the docstring is defined for called __doc__. You can access the attribute using the dot operator:

```
>>> print tools.file_search.__doc__
```

Docstrings are defined for pretty much everything in Python.

```
>>> import math
>>> print math.__doc__
>>> print math.cos.__doc__
>>> print os.__doc__
```

To get even more help, the pydoc module uses the docstring in addition to other documentation to give to detailed information about various python objects.

```
>>> import pydoc
>>> pydoc.help(math)
```

4 Using modules

Modules are great for storing code that you want to reuse, right now, if you want to use a module that you created, it has to be in the directory that you are running Python in. This is not terribly useful.

As such, when Python loads, it looks for the **PYTHONPATH** environment variable that gives the interpreter information on where to look for modules. You can set this variable in your .cshrc file:

```
setenv PYTHONPATH .:'path1':'path2
```

It can be useful to put all of your user defined modules in a directory, and then put that directory in your python path.