Making Your Own Modules

When you're working at the python interpreter, all your code, including the functions you've designed, are lost when you close down the interpreter. So you'll probably want to write all but the smallest snippets of code in files eventually. All python files can be both an executable script and a module, and it is desirable to write your script files in a way that takes advantage of this fact. This cheatsheet will show you how to do this.

Editing PythonPath: This first step is to make sure that you can import your script files from anywhere. To do this, you need to tell Python where to look by editing the global variable $PYTHONPATH. This works the same way as editing your Unix $PATH variable. On Ubuntu, this means adding this line to the bottom of your .bashrc file:

```
export PYTHONPATH=/home/yourname/scripts:$PYTHONPATH
```

Or on Mac:

```
export PYTHONPATH=/home/yourname/scripts:${PYTHONPATH}
```

Name Space: Modules make more sense if you understand a bit about names. A 'name space' is a mapping from a name to an object. For instance, when a function is called, its name is bound to the function object (its 'namespace' is created), and this mapping is then forgotten after execution.

When modules are imported with `import` *`module`*, they define a set of names within the module's name space. Get the full list of names with

```
>>> dir(module)
```

Some of the defined names will be functions within the module that you want to access. Since the function names only have meaning within the imported module's namespace, you have to access them with dot notation:

```
module.function
```

This means that if you import several modules that define functions with the same name, these functions will not get confused with one another. But it also means you can't call the functions by name without the module name first. You can get help on these functions with `help(module.function)`.

When modules are *executed*, rather than imported, their name becomes `'__main__'`

Selective Import: You can call functions from other modules directly by importing these functions by name. Be careful, though: if you import two different functions with the same name, Python won't be able to tell which one you want to use, so do this only when you know all the functions you'll be using.

```
from module import function
from module import *  #imports all functions by name. Be careful!
```

Making Modules: Your scripts are already importable modules. But a few housekeeping tricks make the whole process easier. See the updated version of the obs_counter scripts for an example. The main differences are as follows:

Control Behavior: Adding the `if` clause at the bottom makes it so that certain variables and execution statements are only done when the script itself is executed, rather than imported (i.e. if `__name__` is `'__main__'`). I usually like having a try/except IndexError clause at the bottom which prints out a usage statement if you forget to enter the right number of arguments at the command line when you call this script (which will throw an IndexError).

Passing infile variable: In previous versions, the variable 'infile' was global, meaning any function could access it. Now it's down in the `if` clause at the bottom. This means we have to change the functions so that they take 'infile' as a parameter and pass it up to the function that they call. That way, we can just call print_obs2() at the bottom with 'infile' as a parameter, and it will pass it on up to obs_dictionary(), which will then pass it to opener(), where it is finally used.

Generalize Functions: The point of turning a script into a usable module is so you can access any of its functions and gather their output for some downstream purpose. We've already made a script (obs_counter3.py) which will print out the observations in a nice pretty format, but what if you want to analyze *that* data later? You could run obs_counter3.py into an output file, and then load that file into your downstream code, but it's cleaner to gather that output in background if you're not really interested in it.

To illustrate this, I turned the old function print_obs() into a new function get_obs(). This returns of list where each object in the list is a string, "Organism:*org*\t\tCount:*count*". Here's how to use it

```
   >>> import obs_count_mods
   >>> obs = obs_count_mods.get_obs('file1.txt')
   >>> obs
   ['Organism: Bees\t\tCount: 167', 'Organism: Bird\t\tCount: 58',
'Organism: Bear\t\tCount: 5']
   >>> print obs[0]
   Organism: Bees          Count: 167
```

If I had wanted to analyze this data in another script, it would be easy enough to import obs_count_mods.py into that script and create an 'observations' variable just as I've done above.

Pyc files: After importing one of your scripts, you will notice a new file called `your_module.pyc`. Don't be alarmed! This is a compiled python file. It's written in binary, so you don't want to open it. It's just there so that next time you import your module, python doesn't have to re-compile your original script, making it all go much faster. Just forgettaboutit.

Editing your modules: Making your scripts into modules also allows you to develop your code and do error testing more easily. This is because you can test your functions one-by-one by importing them, which is pretty handy when you have big ol' scripts whose parts you can't remember too well. To effectively test your code by importing them to the interpreter, you'll need one more tool:

```
reload(module)
```

After you've tested a function and changed it in the source code, you'll have to reload the module (after saving, of course) before you can test out the changes you've made.