Learning Python

Getting results for beamlines and scientific programming

Basic Python 9: Creating your own modules

Outline of topics to be covered

1. Why create modules?
2. How to code modules
3. Self-documenting code (pydoc)
4. Incorporating test code
5. How does Python handle modules
6. Examples/Style

Creating your own modules: Why?

When developing software, it is often a good idea to separate code into multiple modules that each implement a collection of related tasks. This offers several advantages:

- **Code reuse**: by creation of a module that does one specific thing, it becomes easier to redeploy that code in a different project should you need to do that again elsewhere.
- **Testing**: for reliable code, one should write functions with supplied tests that exercise all possible things that each function does (best: write the test, before the code).
- **Readability**: a large program is more easily understood when it is written with a simple high-level routine and all the complex "stuff" is isolated in modules

Creating your own modules: More

- **Separation of code**: good programming practice (Model-View-Controller) is to separate modules that implement plotting (view) from the GUI (controller) and from the computations (model)
  - This will help a lot if you want to define a "batch" mechanism for doing the computation or want to move code into a new implementation
- **Code encapsulation:** if one isolates global variables into a module (and better yet uses routines to set and read the variables) it becomes much easier to see where these variables are accessed and changed – this really helps in debugging.

Creating your own modules: How?

Creation of a module is pretty easy. Create a separate .py file with routines and then "import" it where needed. However, a bit of extra work will be worthwhile. Here is a template I recommend following when developing your own module

- Place a docstring at the top that explains what the module does
- Then place all imports at the top, so that it is clear what modules your code required. Good style is to order the modules as:
  - (1) standard;
  - (2) distributed packages;
  - (3) locally-written code
- Define and comment global variables
- For each routine in the module, include a docstring that defines what the routine does, what is expected for the input, what does the function return.

```
"""Docstring that describes
   module"""
import <package>

#global variables

def routine(args1,…):
   """docstring for routine
     arg1 is …
     returns True if…
   """
   ...
   return True

if __name__ == '__main__':
    # example/test code here
```

Following the previous boilerplate creates self-documenting code

```
"""this is file DummyMod.py
example to show how modules are used"""
MyVar = 'Inside '+__name__
def ShowVar():
    "Show the local variable value"
    print "MyVar is",MyVar

ShowVar()

if __name__ == '__main__':
   MyVar = "test variable"
   print "MyVar is",MyVar
```

```
bht3:work toby$ pydoc DummyMod
MyVar is Inside DummyMod
Help on module DummyMod:

NAME
    DummyMod

FILE
    /Users/toby/software/work/DummyMod.py

DESCRIPTION
    this is file DummyMod.py
    example to show how modules are used

FUNCTIONS
    ShowVar()
        Show the local variable value

DATA
    MyVar = 'Inside DummyMod'
```

- Note that even better self-documenting code can be developed using Sphinx, http://sphinx.pocoo.org/
  - generates cross references, web pages, LaTeX…

Creating your own modules: Including test code

The "if __name__ …" statement will evaluate as True when the module is read directly (as in `python file.py`) but will evaluate as False when the module is imported.

- The ideal here is to test every routine against values computed from an external source in a way that every if block (etc.) is exercised.
- In this way, it is easy to check that any change to the code does not break it with respect to previous results
- Even with a minimal testing protocol, the test routines still provide an example for how the code is used.

```
"""Docstring that describes
   module"""
import <package>

# define global variables

def routine(args1,…):
    """docstring for routine
      arg1 is …
      returns True if…
    """
    ...
    return True

if __name__ == '__main__':
    # example/test code here
```

Creating your own modules: Including test code

The "if __name__ …" statement will evaluate as True when the module is read directly (as in `python file.py`) but will evaluate as False when the module is imported.

- The ideal here is to test every routine against values computed from an external source in a way that every if block (etc.) is exercised.
- In this way, it is easy to check that any change to the code does not break it with respect to previous results
- Even with a minimal testing protocol, the test routines still provide an example for how the code is used.

```
"""Docstring that describes
   module"""
import <package>

# define global variables

def routine(args1,…):
    """docstring for routine
      arg1 is …
      returns True if…
    """
    ...
    return True

if __name__ == '__main__':
    # example/test code here
```

How does Python use modules?

- The first time a Python code encounters an `import <module>` statement, the module is located by searching the `sys.path` list for a file named `<module>.pyc or <module>.py` -- the newer one is used
  - The .pyc file is an interpreted version of the .py file – platform independent. It speeds loading but not dramatically (IMHO).
  - The .pyc file updated if needed and allowed.
- The Python code in the module is run the first time the module is imported (but not the "`if __name__ == '__main__':`" section of code, if present). Subsequent import statements will not cause the code to be run.

- Functions are accessed using `<module>.<function>(…)`
- Global variables can be read or changed using `<module>.<var>`

- To force the contents of a module to be rerun use `reload(Mod)` where `Mod` is the variable assigned (`import Mod` or `import Module as Mod`)

Modules inside Modules: hierarchical naming

- If Mod1 imports Mod2:

```
# file Mod1.py
import Mod2
Var = 1
```

```
# file Mod2.py
Var = 2
```

- then one can use the module names hierarchically:

```
import Mod1
Mod1.Mod2.Var = 3
```

The code below prints 3 then 1 since `Mod1.Mod2.Var` and `Mod2.Var` are exactly the same global variable, but Mod1.Var is never changed.

  - Nothing changes if the order of imports is changed

```
import Mod1
Mod1.Mod2.Var = 3
import Mod2
print Mod2.Var
print Mod1.Var
```

Style: Where possible, minimize external global variable use

- For some tasks, global variables are invaluable, but it is more transparent to use function arguments.

```
import mycalc
mycalc.factorialin = 5
factout = mycalc.factorial()
```

```
import mycalc
factout = mycalc.factorial(5)
```

- When a variable will be used in a large number of functions, it does makes sense to implement that with a global variable, but still setting it via a function will be a bit more clear than setting the global variable directly

```
import mycalc
mycalc.DebugLevel = 1
factout = mycalc.factorial(5)
```

```
import mycalc
mycalc.SetDebugLevel(1)
factout = mycalc.factorial(5)
```

Example modules: 1. plotting

see plotnotebook.py file (on http://www.google.com/search?ie=UTF-8&oe=UTF-8&sourceid=navclient&gfns=1&q=pig)

This simplifies creating tabbed plot windows, optionally with wxPython GUI elements. Running the module tests it and shows how to use the code.

Routines:

- Initialization is done when the module is imported.
- function MakePlotWindow() creates a plot window (returns window object).
  - <window>.AddPlotTab(name="name") creates a tab in the plot window, returns the plot object
- function ShowPlots() must be called as the last line in the script.

I will discuss how this works in more detail in the next lecture

Other sample modules

- There is lots of Python code around. Two modules that I have put some time into making "pretty" are:
  - https://subversion.xor.aps.anl.gov/pyGSAS/trunk/GSASIIlattice.py and
  - https://subversion.xor.aps.anl.gov/pyGSAS/trunk/GSASIIspc.py

  Note how unit tests have been provided to make reasonably comprehensive tests of a significant fraction of the routines.

- Google's internal style guide, http://google-styleguide.googlecode.com/svn/trunk/pyguide.html, is worth looking at.

In summary: Create Modules!

- Creation of modules is a good idea. A really good idea!
- As you find the Python file you are working on is getting larger and larger, you should ask yourself “is there a good reason all these routines need to be in a single file?”
- Lots of small modules that each support different types of computations will help you and others reuse code.

Homework, part 1

- The code below is to be placed in a file called DummyMod.py.
  - What happens if the code is invoked directly (`python DummyMod.py`)?
  - What happens if the code is imported (`import DummyMod`)?

```
"""this is file DummyMod.py
example to show how modules are used"""
MyVar = 'Inside '+__name__
def ShowVar():
    "Show the local variable value"
    print "MyVar is",MyVar

ShowVar()

if __name__ == '__main__':
   MyVar = "test variable"
   print "MyVar is",MyVar
```

Homework, part 2

- If we create a 2nd module, a file called DummyMod1.py with this contents:

```
# module DummyMod1.py
import DummyMod
DummyMod.MyVar = 'Inside '+__name__
DummyMod.ShowVar()
```

  - What happens if the code is invoked directly (`python DummyMod.py`)?
  - What happens if the code below is run?
  - Does anything change if DummyMod is imported before DummyMod1 in the code below?

```
import DummyMod1
import DummyMod
DummyMod.MyVar = 'Inside '+__name__
DummyMod.ShowVar()
```