Slides from INF3331 lectures - modules and doc strings

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011

Python modules

Contents

- Making a module
- Making Python aware of modules
- Packages
- Distributing and installing modules

More info

- Appendix B.1 in the course book
- Python electronic documentation:
  Distributing Python Modules, Installing Python Modules

Make your own Python modules!

- Reuse scripts by wrapping them in classes or functions
- Collect classes and functions in library modules
- How? just put classes and functions in a file MyMod.py
- Put MyMod.py in one of the directories where Python can find it (see next slide)
- Say

```
import MyMod
# or
import MyMod as M   # M is a short form
# or
from MyMod import *
# or
from MyMod import myspecialfunction, myotherspecialfunction
```

in any script

How Python can find your modules

- Python has some 'official' module directories, typically

  ```
  /usr/lib/python2.3
  /usr/lib/python2.3/site-packages
  ```

  + current working directory

- The environment variable PYTHONPATH may contain additional directories with modules

  ```
  unix> echo $PYTHONPATH
  /home/me/python/mymodules:/usr/lib/python2.2:/home/you/yourlib
  ```

- Python's `sys.path` list contains the directories where Python searches for modules
- `sys.path` contains 'official' directories, plus those in PYTHONPATH)

Setting PYTHONPATH

- In a Unix Bash environment environment variables are normally set in `.bashrc`:

```
export PYTHONTPATH=$HOME/pylib:$scripting/src/tools
```

- Check the contents:

```
unix> echo $PYTHONPATH
```

- In a Windows environment one can do the same in `autoexec.bat`:

```
set PYTHONPATH=C:\pylib;%scripting%\src\tools
```

- Check the contents:

```
dos> echo %PYTHONPATH%
```

- Note: it is easy to make mistakes; PYTHONPATH may be different from what you think, so check `sys.path`

Summary of finding modules

- Copy your module file(s) to a directory already contained in `sys.path`

  ```
  unix or dos> python -c 'import sys; print sys.path'
  ```

- Can extend PYTHONPATH

  ```
  # Bash syntax:
  export PYTHONPATH=$PYTHONPATH:/home/me/python/mymodules
  ```

- Can extend `sys.path` in the script:

  ```
  sys.path.insert(0, '/home/me/python/mynewmodules')
  ```

  (insert first in the list)

Packages (1)

- A class of modules can be collected in a *package*
- Normally, a package is organized as module files in a directory tree
- Each subdirectory has a file `__init__.py`
  (can be empty)
- Packages allow “dotted modules names” like

  `MyMod.numerics.pde.grids`

  reflecting a file `MyMod/numerics/pde/grids.py`

Packages (2)

- Can import modules in the tree like this:

```
from MyMod.numerics.pde.grids import fdm_grids

grid = fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
...
```

  Here, class `fdm_grids` is in module `grids` (file `grids.py`) in the directory `MyMod/numerics/pde`

- Or

```
import MyMod.numerics.pde.grids
grid = MyMod.numerics.pde.grids.fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
#or
import MyMod.numerics.pde.grids as Grid
grid = Grid.fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
```

- See ch. 6 of the Python Tutorial (part of the electronic doc)

Test/doc part of a module

- Module files can have a test/demo script at the end:

```
if __name__ == '__main__':
    infile = sys.argv[1]; outfile = sys.argv[2]
    for i in sys.argv[3:]:
        create(infile, outfile, i)
```

- The block is executed if the module file is run as a script
- The tests at the end of a module often serve as good examples on the usage of the module

Public/non-public module variables

- Python convention: add a leading underscore to non-public functions and (module) variables

```
_counter = 0

def _filename():
    """Generate a random filename."""
    ...
```

- After a standard import `import MyMod`, we may access

```
MyMod._counter
n = MyMod._filename()
```

  but after a `from MyMod import *` the names with leading underscore are *not* available

- Use the underscore to tell users what is public and what is not
- Note: non-public parts can be changed in future releases

Installation of modules/packages

- Python has its own build/installation system: Distutils
- Build: compile (Fortran, C, C++) into module
  (only needed when modules employ compiled code)
- Installation: copy module files to “install” directories
- Publish: make module available for others through PyPi
- Default installation directory:

```
os.path.join(sys.prefix, 'lib', 'python' + sys.version[0:3],
             'site-packages')
# e.g. /usr/lib/python2.3/site-packages
```

- Distutils relies on a `setup.py` script

A simple setup.py script

- Say we want to distribute two modules in two files

```
MyMod.py  mymodcore.py
```

- Typical `setup.py` script for this case:

```
#!/usr/bin/env python
from distutils.core import setup

setup(name='MyMod',
      version='1.0',
      description='Python module example',
      author='Hans Petter Langtangen',
      author_email='hpl@ifi.uio.no',
      url='http://www.simula.no/pymod/MyMod',
      py_modules=['MyMod', 'mymodcore'],
     )
```

setup.py with compiled code

- Modules can also make use of Fortran, C, C++ code
- `setup.py` can also list C and C++ files; these will be compiled with the same options/compiler as used for Python itself
- SciPy has an extension of Distutils for “intelligent” compilation of Fortran files
- Note: `setup.py` eliminates the need for makefiles
- Examples of such `setup.py` files are provided in the section on mixing Python with Fortran, C and C++

Installing modules

- Standard command:

```
python setup.py install
```

- If the module contains files to be compiled, a two-step procedure can be invoked

```
python setup.py build
# compiled files and modules are made in subdir. build/
python setup.py install
```

Controlling the installation destination

- `setup.py` has many options
- Control the destination directory for installation:

```
python setup.py install --prefix=$HOME/install
# copies modules to /home/hpl/install/lib/python
```

- Make sure that `/home/hpl/install/lib/python` is registered in your `PYTHONPATH`

How to learn more about Distutils

- Go to the official electronic Python documentation
- Look up “Distributing Python Modules”
  (for packing modules in `setup.py` scripts)
- Look up “Installing Python Modules”
  (for running `setup.py` with various options)

Doc strings

Contents

- How to document *usage* of Python functions, classes, modules
- Automatic testing of code (through doc strings)

More info

- App. B.1/B.2 in the course book
- HappyDoc, Pydoc, Epydoc manuals
- Style guide for doc strings (see `doc.html`)

Doc strings (1)

- Doc strings = first string in functions, classes, files
- Put user information in doc strings:

```
def ignorecase_sort(a, b):
    """Compare strings a and b, ignoring case."""
    ...
```

- The doc string is available at run time and explains the purpose and usage of the function:

```
>>> print ignorecase_sort.__doc__
'Compare strings a and b, ignoring case.'
```

Doc strings (2)

- Doc string in a class:

```
class MyClass:
    """Fake class just for exemplifying doc strings."""

    def __init__(self):
        ...
```

- Doc strings in modules are a (often multi-line) string starting in the top of the file

```
"""
This module is a fake module
for exemplifying multi-line
doc strings.
"""
```

Doc strings (3)

- The doc string serves two purposes:
  - documentation in the source code
  - on-line documentation through the attribute `__doc__`
  - documentation generated by, e.g., HappyDoc
- HappyDoc: Tool that can extract doc strings and automatically produce overview of Python classes, functions etc.
- Doc strings can, e.g., be used as balloon help in sophisticated GUIs (cf. IDLE)
- Providing doc strings is a good habit!

Doc strings (4)

There is an official style guide for doc strings:

- PEP 257 "Docstring Conventions" from http://www.python.org/dev/peps/
- Use triple double quoted strings as doc strings
- Use complete sentences, ending in a period

```
def somefunc(a, b):
    """Compare a and b."""
```

Automatic doc string testing (1)

- The `doctest` module enables automatic testing of interactive Python sessions embedded in doc strings

```
class StringFunction:
    """
    Make a string expression behave as a Python function
    of one variable.
    Examples on usage:
    >>> from StringFunction import StringFunction
    >>> f = StringFunction('sin(3*x) + log(1+x)')
    >>> p = 2.0; v = f(p)  # evaluate function
    >>> p, v
    (2.0, 0.8191967904691839 2)
    >>> f = StringFunction('1+t', independent_variables='t')
    >>> v = f(1.2)  # evaluate function of t=1.2
    >>> print "%.2f" % v
    2.20
    >>> f = StringFunction('sin(t)')
    >>> v = f(1.2)  # evaluate function of t=1.2
    Traceback (most recent call last):
        v = f(1.2)
    NameError: name 't' is not defined
    """
```

Automatic doc string testing (2)

- Class `StringFunction` is contained in the module `StringFunction`
- Let `StringFunction.py` execute two statements when run as a script:

```
def _test():
    import doctest
    return doctest.testmod(StringFunction)

if __name__ == '__main__':
    _test()
```

- Run the test:

```
python StringFunction.py       # no output: all tests passed
python StringFunction.py  -v   # verbose output
```