1.2.5. Reusing code: scripts and modules

For now, we have typed all instructions in the interpreter. For longer sets of instructions we need to change track and write the code in text files (using a text editor), that we will call either *scripts* or *modules*. Use your favorite text editor (provided it offers syntax highlighting for Python), or the editor that comes with the Scientific Python Suite you may be using (e.g., Scite with Python(x,y)).

1.2.5.1. Scripts

Let us first write a script, that is a file with a...

The extension for Python files is `.py`. Write or copy-and-paste the following lines in a file called `test.py`

```
message = "Hello how are you?"
for word in message.split():
    print word
```

Let us now execute the script interactively, that is inside the Ipython interpreter. This is maybe the most common use of scripts in scientific computing.

Note: in Ipython, the syntax to execute a script is `%run script.py`. For example,

```
In [1]: %run test.py
Hello
how
are
```

you?

```
In [2]: message
Out[2]: 'Hello how are you?'
```

The script has been executed. Moreover the variables defined in the script (such as `message`) are now available inside the interpreter's namespace.

Other interpreters also offer the possibility to execute scripts (e.g., `execfile` in the plain Python interpreter, etc.).

It is also possible In order to execute this script as a *standalone program*, by executing the script inside a shell terminal (Linux/Mac console or cmd Windows console). For example, if we are in the same directory as the test.py file, we can execute this in a console:

```
$ python test.py
Hello
how
are
you?
```

Standalone scripts may also take command-line arguments

In `file.py`:

```
import sys
print sys.argv
```

```
$ python file.py test arguments
['file.py', 'test', 'arguments']
```

⚠ Don't implement option parsing yourself. Use modules such as `optparse`, `argparse` or `docopt`.

1.2.5.2. Importing objects from modules

```
In [1]: import os

In [2]: os
Out[2]: <module 'os' from '/usr/lib/python2.6/os.pyc'>

In [3]: os.listdir('.')
Out[3]:
['conf.py',
 'basic_types.rst',
 'control_flow.rst',
 'functions.rst',
 'python_language.rst',
 'reusing.rst',
 'file_io.rst',
 'exceptions.rst',
 'workflow.rst',
 'index.rst']
```

And also:

```
In [4]: from os import listdir
```

Importing shorthands:

```
In [5]: import numpy as np
```

⚠

```
from os import *
```

This is called the *star import* and please, **Use it with caution**

- Makes the code harder to read and understand: where do symbols come from?
- Makes it impossible to guess the functionality by the context and the name (hint: *os.name* is the name of the OS), and to profit usefully from tab completion.
- Restricts the variable names you can use: *os.name* might override *name*, or vise-versa.
- Creates possible name clashes between modules.
- Makes the code impossible to statically check for undefined symbols.

Modules are thus a good way to organize code in a hierarchical way. Actually, all the scientific computing tools we are going to use are modules:

```
>>> import numpy as np # data arrays
>>> np.linspace(0, 10, 6)
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> import scipy # scientific computing
```

In Python(x,y), Ipython(x,y) executes the following imports at startup:

```
>>> import numpy
>>> import numpy as np
>>> from pylab import *
>>> import scipy
```

and it is not necessary to re-import these modules.

1.2.5.3. Creating modules

If we want to write larger and better organized programs (compared to simple scripts), where some objects are defined, (variables, functions, classes) and that we want to reuse several times, we have to create our own *modules*.

Let us create a module `demo` contained in the file `demo.py`:

```
"A demo module."

def print_b():
    "Prints b."
    print 'b'

def print_a():
    "Prints a."
    print 'a'


c = 2
d = 2
```

In this file, we defined two functions `print_a` and `print_b`. Suppose we want to call the `print_a` function from the interpreter. We could execute the file as a script, but since we just want to have access to the function `print_a`, we are rather going to **import it as a module**. The syntax is as follows.

```
In [1]: import demo


In [2]: demo.print_a()
a

In [3]: demo.print_b()
b
```

Importing the module gives access to its objects, using the `module.object` syntax. Don't forget to put the module's name before the object's name, otherwise Python won't recognize the instruction.

Introspection

```
In [4]: demo?
Type:               module
Base Class: <type 'module'>
String Form:        <module 'demo' from 'demo.py'>
Namespace:  Interactive
File:
      /home/varoquau/Projects/Python_talks/scipy_2009_tutorial/source/demo.p
      y
Docstring:
    A demo module.


In [5]: who
demo

In [6]: whos
Variable   Type      Data/Info
------------------------------
demo       module    <module 'demo' from 'demo.py'>

In [7]: dir(demo)
Out[7]:
['__builtins__',
'__doc__',
'__file__',
'__name__',
'__package__',
'c',
'd',
'print_a',
'print_b']


In [8]: demo.
```

```
demo.__builtins__      demo.__init__          demo.__str__
demo.__class__         demo.__name__          demo.__subclasshook__
demo.__delattr__       demo.__new__           demo.c
demo.__dict__          demo.__package__       demo.d
demo.__doc__           demo.__reduce__        demo.print_a
demo.__file__          demo.__reduce_ex__     demo.print_b
demo.__format__        demo.__repr__          demo.py
demo.__getattribute__  demo.__setattr__       demo.pyc
demo.__hash__          demo.__sizeof__
```

Importing objects from modules into the main namespace

```
In [9]: from demo import print_a, print_b

In [10]: whos
Variable   Type        Data/Info
------------------------------
demo       module      <module 'demo' from 'demo.py'>
print_a    function    <function print_a at 0xb7421534>
print_b    function    <function print_b at 0xb74214c4>

In [11]: print_a()
a
```

Module caching

Modules are cached: if you modify `demo.py` and re-import it in the old session, you will get the old one.

Solution:

```
In [10]: reload(demo)
```

In Python3 instead `reload` is not builtin, so you have to import the `importlib` module first and then do:

```
In [10]: importlib.reload(demo)
```

1.2.5.4. '__main__' and module loading

Sometimes we want code to be executed when a module is run directly, but not when it is imported by another module. `if __name__ == '__main__'` allows us to check whether the module is being run directly.

File `demo2.py`:

```
def print_b():
    "Prints b."
    print 'b'

def print_a():
    "Prints a."
    print 'a'

# print_b() runs on import
print_b()

if __name__ == '__main__':
    # print_a() is only executed when the module is run directly.
    print_a()
```

Importing it:

```
In [11]: import demo2
b
```

```
In [12]: import demo2
```

Running it:

```
In [13]: %run demo2
b
a
```

1.2.5.5. Scripts or modules? How to organize your code

Note: Rule of thumb

- Sets of instructions that are called several times should be written inside **functions** for better code reusability.
- Functions (or other bits of code) that are called from several scripts should be written inside a **module**, so that only the module is imported in the different scripts (do not copy-and-paste your functions in the different scripts!).

1.2.5.5.1. How modules are found and imported

When the `import mymodule` statement is executed, the module `mymodule` is searched in a given list of directories. This list includes a list of installation-dependent default path (e.g., `/usr/lib/python`) as well as the list of directories specified by the environment variable `PYTHONPATH`.

The list of directories searched by Python is given by the `sys.path` variable

```
In [1]: import sys

In [2]: sys.path
```

```
Out[2]:
['',
 '/home/varoquau/.local/bin',
 '/usr/lib/python2.7',
 '/home/varoquau/.local/lib/python2.7/site-packages',
 '/usr/lib/python2.7/dist-packages',
 '/usr/local/lib/python2.7/dist-packages',
 ...]
```

Modules must be located in the search path, therefore you can:

- write your own modules within directories already defined in the search path (e.g. `$HOME/.local/lib/python2.7/dist-packages`). You may use symbolic links (on Linux) to keep the code somewhere else.
- modify the environment variable `PYTHONPATH` to include the directories containing the user-defined modules.

  On Linux/Unix, add the following line to a file read by the shell at startup (e.g. /etc/profile, .profile)

  ```
  export PYTHONPATH=$PYTHONPATH:/home/emma/user_defined_modules
  ```

  On Windows, http://support.microsoft.com/kb/310519 explains how to handle environment variables.

- or modify the `sys.path` variable itself within a Python script.

  ```
  import sys
  new_path = '/home/emma/user_defined_modules'
  if new_path not in sys.path:
      sys.path.append(new_path)
  ```

  This method is not very robust, however, because it makes the code less portable (user-dependent path) and because you have to add the directory to your sys.path each time you want to import from a module in this directory.

See also: See https://docs.python.org/tutorial/modules.html for more information about modules.

1.2.5.6. Packages

A directory that contains many modules is called a *package*. A package is a module with submodules (which can have submodules themselves, etc.). A special file called `__init__.py` (which may be empty) tells Python that the directory is a Python package, from which modules can be imported.

```
$ ls
cluster/        io/           README.txt@     stsci/
__config__.py@  LATEST.txt@   setup.py@       __svn_version__.py@
__config__.pyc  lib/          setup.pyc       __svn_version__.pyc
constants/      linalg/       setupscons.py@  THANKS.txt@
fftpack/        linsolve/     setupscons.pyc  TOCHANGE.txt@
__init__.py@    maxentropy/   signal/         version.py@
__init__.pyc    misc/         sparse/         version.pyc
INSTALL.txt@    ndimage/      spatial/        weave/
integrate/      odr/          special/
interpolate/    optimize/     stats/
$ cd ndimage
$ ls
doccer.py@   fourier.pyc   interpolation.py@  morphology.pyc   setup.pyc
doccer.pyc   info.py@      interpolation.pyc  _nd_image.so
setupscons.py@
filters.py@  info.pyc      measurements.py@   _ni_support.py@
setupscons.pyc
filters.pyc  __init__.py@  measurements.pyc   _ni_support.pyc  tests/
fourier.py@  __init__.pyc  morphology.py@     setup.py@
```

From Ipython:

```
In [1]: import scipy

In [2]: scipy.__file__
Out[2]: '/usr/lib/python2.6/dist-packages/scipy/__init__.pyc'

In [3]: import scipy.version

In [4]: scipy.version.version
Out[4]: '0.7.0'

In [5]: import scipy.ndimage.morphology

In [6]: from scipy.ndimage import morphology

In [17]: morphology.binary_dilation?
Type:           function
Base Class:     <type 'function'>
String Form:    <function binary_dilation at 0x9bedd84>
Namespace:      Interactive
File:           /usr/lib/python2.6/dist-packages/scipy/ndimage/morphology.py
Definition:     morphology.binary_dilation(input, structure=None,
iterations=1, mask=None, output=None, border_value=0, origin=0,
brute_force=False)
Docstring:
    Multi-dimensional binary dilation with the given structure.

    An output array can optionally be provided. The origin parameter
    controls the placement of the filter. If no structuring element is
    provided an element is generated with a squared connectivity equal
    to one. The dilation operation is repeated iterations times.  If
    iterations is less than 1, the dilation is repeated until the
    result does not change anymore.  If a mask is given, only those
    elements with a true value at the corresponding mask element are
    modified at each iteration.
```

1.2.5.7. Good practices

- Use **meaningful** object **names**
- **Indentation: no choice!**

  Indenting is compulsory in Python! Every command block following a colon bears an additional indentation level with respect to the previous line with a colon. One must therefore indent after `def f():` or `while:`. At the end of such logical blocks, one decreases the indentation depth (and re-increases it if a new block is entered, etc.)

  Strict respect of indentation is the price to pay for getting rid of `{` or `;` characters that delineate logical blocks in other languages. Improper indentation leads to errors such as

```
------------------------------------------------------------
IndentationError: unexpected indent (test.py, line 2)
```

  All this indentation business can be a bit confusing in the beginning. However, with the clear indentation, and in the absence of extra characters, the resulting code is very nice to read compared to other languages.

- **Indentation depth**: Inside your text editor, you may choose to indent with any positive number of spaces (1, 2, 3, 4, ...). However, it is considered good practice to **indent with 4 spaces**. You may configure your editor to map the `Tab` key to a 4-space indentation. In Python(x,y), the editor is already configured this way.
- **Style guidelines**

  Long lines: you should not write very long lines that span over more than (e.g.) 80 characters. Long lines can be broken with the `\` character

```
>>> long_line = "Here is a very very long line \
... that we break in two parts."
```

  Spaces

Write well-spaced code: put whitespaces after commas, around arithmetic operators, etc.:

```
>>> a = 1 # yes
>>> a=1 # too cramped
```

A certain number of rules for writing “beautiful” code (and more importantly using the same conventions as anybody else!) are given in the Style Guide for Python Code.

Quick read

If you want to do a first quick pass through the Scipy lectures to learn the ecosystem, you can directly skip to the next chapter: NumPy: creating and manipulating numerical data.

The remainder of this chapter is not necessary to follow the rest of the intro part. But be sure to come back and finish this chapter later.
