Coding Standards - Team 8

Naming conventions

All new IPython code (and much existing code is being refactored), we'll use:

- All lowercase module names.
- CamelCase for class names.
- lowercase_with_underscores for methods, functions, variables and attributes.

When subclassing classes that use other naming conventions, you must follow their naming conventions. To deal with cases like this, we propose the following policy:

- If you are subclassing a class that uses different conventions, use its naming conventions throughout your subclass. Thus, if you are creating a Twisted Protocol class, used Twisted's namingSchemeForMethodsAndAttributes.
- All IPython's official interfaces should use our conventions. In some cases this will mean that you need to provide shadow names (first implement fooBar and then foo_bar = fooBar).

Implementation-specific *private* methods will use _single_underscore_prefix. Names with a leading double underscore will *only* be used in special cases, as they makes subclassing difficult (such names are not easily seen by child classes).

Occasionally some run-in lowercase names are used, but mostly for very short names or where we are implementing methods very similar to existing ones in a base class (like runlines() where runsource() and runcode() had established precedent).

Attribute declarations for objects

By declaring all attributes of the object in the class header, there is a single place one can refer to for understanding the object's data interface, where comments can explain the role of each variable and when possible, sensible deafaults can be assigned.

If an attribute is meant to contain a mutable object, it should be set to None in the class and its mutable value should be set in the object's constructor. Since class attributes are shared by all instances, failure to do this can lead to difficult to track bugs. But you should still set it in the class declaration so the interface specification is complete and documdented in one place.

A simple example:

```
class foo:
  # X does..., sensible default given:
  x = 1
  # y does..., default will be set by constructor
```

```
    y = None
    # z starts as an empty list, must be set in constructor
    z = None

    def __init__(self, y):
        self.y = y
        self.z = []
```

New files

When starting a new file for IPython, you can use the following template as a starting point that has a few common things pre-written for you. The template is included in the documentation sources as docs/sources/development/template.py:

```
# Imports

from __future__ import print_function

# [remove this comment in production]

# List all imports, sorted within each section (stdlib/third-party/ipython).

# For 'import foo', use one import per line.  For 'from foo.bar import a, b, c'

# it's OK to import multiple items, use the parenthesized syntax 'from foo

# import (a, b, ...)' if the list needs multiple lines.

# Stdlib imports

# Third-party imports

# Our own imports

# Use broad section headers like this one that make it easier to navigate the

# file, with descriptive titles.  For complex classes, simliar (but indented)

# headers are useful to organize the internal class structure.

# Globals and constants

# Local utilities

# Classes and functions
```