**Python Coding Style & Standards**

[**Introduction**](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id1)

This of course is partly due to the excellent, and very sound, Python bible of programming style: [PEP 8](http://www.python.org/peps/pep-0008.html). There is also a sensible list of Python coding style guidelines at Python Idioms and Efficiency Suggestions. Personally I find it slightly painful to read code that deviates too far from these standards, but always bearing in mind that a foolish consistency is the hobgoblin of little mind.

**Standards and Style**

Python uses indentation to give you a visual indication of the structure of your code. Additionally, we have an interactive interpreter which provides a default representation for many objects and data structures. Consistency of style, and using the interpreters representation for objects, will make it easier for you to parse and absorb the meaning of the code.

Python is a beautiful language, so keep your source code beautiful too...

## Four Space Indentation

And definitely no tabs.

Docstris

I prefer docstrings with either of the following two styles:

def function\_class\_or\_method(object):  
    """A single line docstring."""

or

def function\_class\_or\_method(object):  
    """  
    Several lines of docstring.  
  
    Preferably using ReST 'markup'.  
    """

## [Line Length](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id5)

Over long lines hurt readability, but so does breaking lines purely because of setting an arbitrary maximum line length. A limit of 79 characters is a good guideline, but readability should come first.

Longer lines should be wrapped by surrounding expressions in parentheses rather than using \. Long strings can be split across several lines using parentheses:

('This is an excessively long string. '  
'So, what we can do is split it across, '  
'several %s. Instead of one long one. Good hey?' %  
'lines')

Note that the single value for string formatting doesn't have a trailing comma to turn it into a tuple (a one-ple?). They annoy me. (But they can be necessary if the object you are interpolating is itself a tuple.)

I also prefer single quotes to double quotes, however I prefer double quotes to escaped single quotes. (We have both quote marks; we might as well use them. It's not as if we need to persuade some compiler that the string that follows really is more than one-character long.)

Where long expressions go over more than one line, it is helpful to indent lines after the first one. This shows that they belong to each other.

result = (question1() + question2() + question3() +  
          question4() + question5() + 27)

If possible the indented lines should all line up. For multiline containers the following style is fine though:

values = [  
    value1,  
    value2,  
    value3  
]

## [Line Spacing](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id6)

Functions and methods should be separated with two blank lines. Class definitions with three blank lines.

Normally variable definitions don't need a blank line between them, unless they are logically distinct and you want to separate them. Logical chunks of code within function/method definitions can have a blank line if you want to visually separate them. With these rules the blank lines provide a visual guide to the structure of the code.

(I often omit the blank lines between the class decoration and \_\_init\_\_.)

"""Module docstring."""  
  
var1 = 'Some Value'  
var2 = "Another value."  
  
CONST1 = 'Important constant'  
CONST2 = 'Another constant'  
  
  
class ClassName(object):  
    """Class docstring."""  
  
    class\_attribute = 3  
  
  
    def \_\_init\_\_(self):  
        """Method docstring."""  
        pass  
  
  
    def another\_method(self):  
        """Another docstring."""  
        print 'A method.'  
        #  
        x = 3  
  
  
  
class AnotherClass(object):

## [Comments](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id7)

No inline comments, comments should be above the line they comment.

It can be useful to precede comments with FIXME: or NOTE: if it clarifies the intent of the comment.

Since I started practising TDD, I believe that comments should be used as sparingly as possible. Code should be self-documenting as much as possible (clarified by test cases), and comments should be used to explain unavoidable complexity or obscurity.

## [Parentheses Round Expressions](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id8)

Some expressions can get complicated. Parentheses can (and should) be used to make them less ambiguous. This is for the sake of people who read the code, even if it doesn't matter to the Python parser.

if (test1 and test2) or (test3 or test4):

## [Object Representations](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id9)

Where you use lambdas and dictionaries in code, they should use the same style as their repr (note the lack of spaces before the colons):

lambda: somefunction(3)  
lambda x: function(x)  
  
dictionary = {'a': 1, 'b': 2}

## [Programming Rules](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id10)

These few rules apply to how you code, rather than just style.

* When testing for None, use is.

if obj is None

This is because None is a singleton and the identity test is more efficient than testing for equality. Where what you really mean is an identity rather than equality test then is can also make your code more clearly convey its intent.

* Classes should inherit from object.

If a class has no base classes, then it is better to make it a new style class and inherit from object.

class Something(object):

* No mutable objects as default arguments.

Default arguments are created on parsing, not when a function/method is called. This means you mustn't use mutable objects (like dictionaries or lists) as default arguments.

Instead you should do :

def function(default=None):  
    if default is None:  
        default = {}

* No bare excepts. If you really want to catch all exceptions, then use except Exception:, but it is many times better to only trap exceptions that you expect. Doing otherwise will suppress a multitude of bugs.
* If you are writing a module, always define \_\_all\_\_.

## [Multiple Statements per Line](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id11)

Don't do it! Python indentation gives you a visual indication of structure; putting multiple statements per line breaks this. I find all of the following ugly.

if x: print y  
  
a = f(x); print a  
  
try: dosomething()  
except: dosomething\_else()

## [Imports](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id12)

A few simple rules :

* Imports at the top of modules (or the top of functions if they are local).
* Never use from xxx import \*.

Even for importing a lot of names it is better to be able to see where your names come from. Tools like pylint and PyFlakes can help warn you about unused imports.

* I think it looks nicer to put imports on separate lines.

import xxx should come before from xxx import ....

Standard library before extension modules.

* A single module import per line.

I now prefer this:

import os  
import sys

to this:

import os, sys

Not a big deal though.

* Where you are importing lots of names, and targeting Python 2.4 or more recent, you can use the following syntax:

from namespace import (  
    name1, name2, name3,  
    name4, name5, name6  
)

## [Naming Conventions](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id13)

There are various Python naming conventions I use. Consistency here is certainly good as it helps to identify what sort of object names point to. I think the conventions I use basically follow PEP8.

* Module names should be lowercase with underscores instead of spaces. (And should be valid module names for importing.)
* Variable names and function/method names should also be lowercase with underscores to separate words.
* Class names should be CamelCase (uppercase letter to start with, words run together, each starting with an uppercase letter).
* Module constants should be all uppercase.

E.g. You would typically have module.ClassName.method\_name.

Module names in CamelCase with a main class name identical to the module name are annoying. (e.g. ConfigParser.ConfigParser, which should always be spelt configobj.ConfigObj.)

Also, variables, functions, methods and classes which aren't part of your public API, should begin with a single underscore. (using double underscores to make attributes private almost always turns out to be a mistake - especially for testability.)

## [Whitespace](http://www.voidspace.org.uk/python/articles/python_style_guide.shtml#id14)

And finally, you should always have whitespace around operators and after punctuation. The exception is default arguments to methods and functions.

E.g. def function(default=argument): and x = a \* b + c