Python Coding Standard

**Indentation**

Use 4 spaces per indentation level.

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent [7]. When using a hanging indent the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

Yes:

*# Aligned with opening delimiter.*

*foo = long\_function\_name(var\_one, var\_two,*

*var\_three, var\_four)*

*# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.*

*def long\_function\_name(*

*var\_one, var\_two, var\_three,*

*var\_four):*

*print(var\_one)*

*# Hanging indents should add a level.*

*foo = long\_function\_name(*

*var\_one, var\_two,*

*var\_three, var\_four)*

We will use it.

**Tabs or Spaces?**

Spaces are the preferred indentation method. Tabs should be used solely to remain consistent with code that is already indented with tabs. Python 3 disallows mixing the use of tabs and spaces for indentation.

**Source File Encoding**

For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the Latin alphabet (latin-1, ISO/IEC 8859-1 character set) MUST provide a transliteration of their names in this character set. Open source projects with a global audience are encouraged to adopt a similar policy.

**Imports**

* Imports should usually be on separate lines:

*Yes: import os*

*import sys*

* Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
* Imports should be grouped in the following order:
  + Standard library imports.
  + Related third party imports.
  + Local application/library specific imports.
  + You should put a blank line between each group of imports.

**Whitespace in Expressions and Statements**

**Pet Peeves**

Avoid extraneous whitespace in the following situations:

* Immediately inside parentheses, brackets or braces.

*Yes: spam(ham[1], {eggs: 2})*

*No: spam( ham[ 1 ], { eggs: 2 } )*

* Immediately before a comma, semicolon, or colon:

*Yes: if x == 4: print x, y; x, y = y, x*

*No: if x == 4 : print x , y ; x , y = y , x*

* More than one space around an assignment (or other) operator to align it with another.

*Yes:*

*x = 1*

*y = 2*

*long\_variable = 3*

**Other Recommendations**

* Don't use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a default value for an unannotated function parameter.

*Yes:*

*def complex(real, imag=0.0):*

*return magic(r=real, i=imag)*

*No:*

*def complex(real, imag = 0.0):*

*return magic(r = real, i = imag)*

* Compound statements (multiple statements on the same line) are generally discouraged.

*Yes:*

*if foo == 'blah':*

*do\_blah\_thing()*

*do\_one()*

*do\_two()*

*do\_three()*

* When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments or imported items is expected to be extended over time. The pattern is to put each value (etc.) on a line by itself, always adding a trailing comma, and add the close parenthesis/bracket/brace on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter (except in the above case of singleton tuples).

*Yes:*

*FILES = [*

*'setup.cfg',*

*'tox.ini',*

*]*

*initialize(FILES,*

*error=True,*

*)*

**Block Comments**

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line containing a single #.

**Documentation Strings**

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257.

* Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.
* PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself:

*"""Return a foobang*

*Optional plotz says to frobnicate the bizbaz first.*

*"""*

**Naming Conventions**

**Overriding Principle**

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

*Names to Avoid*

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

* \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes \_FooBar\_\_boo; see below).
* \_\_double\_leading\_and\_trailing\_underscore\_\_: "magic" objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_, \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented.

**Class Names**

Class names should normally use the CapWords convention.

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

**Function and Variable Names**

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility

.

**Function and Method Arguments**

* Always use self for the first argument to instance methods.
* Always use cls for the first argument to class methods.
* If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class\_ is better than clss. (Perhaps better is to avoid such clashes by using a synonym.)

**Method Names and Instance Variables**

* Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.
* Use one leading underscore only for non-public methods and instance variables.
* To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

**Designing for Inheritance**

* Public attributes should have no leading underscores.
* If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling.
* If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

**Programming Recommendations**

* Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).
* Use is not operator rather than not ... is. While both expressions are functionally identical, the former is more readable and preferred.

*Yes:*

*if foo is not None:*

*When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause:*

*try:*

*import platform\_specific\_module*

*except ImportError:*

*platform\_specific\_module = None*

* Use string methods instead of the string module.
* Don't compare boolean values to True or False using ==.

*Yes: if greeting:*

*No: if greeting == True:*

*Worse: if greeting is True:*