|  |  |
| --- | --- |
| **PEP:** | 8 |
| **Title:** | Style Guide for Python Code |
| **Version:** | 68852 |
| **Last-Modified:** | [2009-01-22 09:36:39 +0100 (Thu, 22 Jan 2009)](http://svn.python.org/view/*checkout*/peps/trunk/pep-0008.txt) |
| **Author:** | Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org> |
| **Status:** | Active |
| **Type:** | Process |
| **Created:** | 05-Jul-2001 |
| **Post-History:** | 05-Jul-2001 |

**Introduction**

This document gives coding conventions for the Python code comprising the

standard library in the main Python distribution. Please see the

companion informational PEP describing style guidelines for the C code in

the C implementation of Python[1].

This document was adapted from Guido's original Python Style Guide

essay[2], with some additions from Barry's style guide[5]. Where there's

conflict, Guido's style rules for the purposes of this PEP. This PEP may

still be incomplete (in fact, it may never be finished <wink>).

**A Foolish Consistency is the Hobgoblin of Little Minds**

One of Guido's key insights is that code is read much more often than it

is written. The guidelines provided here are intended to improve the

readability of code and make it consistent across the wide spectrum of

Python code. As [PEP 20](http://www.python.org/dev/peps/pep-0020) [6] says, "Readability counts".

A style guide is about consistency. Consistency with this style guide is

important. Consistency within a project is more important. Consistency

within one module or function is most important.

But most importantly: know when to be inconsistent -- sometimes the style

guide just doesn't apply. When in doubt, use your best judgment. Look

at other examples and decide what looks best. And don't hesitate to ask!

Two good reasons to break a particular rule:

(1) When applying the rule would make the code less readable, even for

someone who is used to reading code that follows the rules.

(2) To be consistent with surrounding code that also breaks it (maybe for

historic reasons) -- although this is also an opportunity to clean up

someone else's mess (in true XP style).

**Code lay-out**

Indentation

Use 4 spaces per indentation level.

For really old code that you don't want to mess up, you can continue to

use 8-space tabs.

Tabs or Spaces?

Never mix tabs and spaces.

The most popular way of indenting Python is with spaces only. The

second-most popular way is with tabs only. Code indented with a mixture

of tabs and spaces should be converted to using spaces exclusively. When

invoking the Python command line interpreter with the -t option, it issues

warnings about code that illegally mixes tabs and spaces. When using -tt

these warnings become errors. These options are highly recommended!

For new projects, spaces-only are strongly recommended over tabs. Most

editors have features that make this easy to do.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

There are still many devices around that are limited to 80 character

lines; plus, limiting windows to 80 characters makes it possible to have

several windows side-by-side. The default wrapping on such devices

disrupts the visual structure of the code, making it more difficult to

understand. Therefore, please limit all lines to a maximum of 79

characters. For flowing long blocks of text (docstrings or comments),

limiting the length to 72 characters is recommended.

The preferred way of wrapping long lines is by using Python's implied line

continuation inside parentheses, brackets and braces. If necessary, you

can add an extra pair of parentheses around an expression, but sometimes

using a backslash looks better. Make sure to indent the continued line

appropriately. The preferred place to break around a binary operator is

\*after\* the operator, not before it. Some examples:

class Rectangle(Blob):

def \_\_init\_\_(self, width, height,

color='black', emphasis=None, highlight=0):

if width == 0 and height == 0 and \

color == 'red' and emphasis == 'strong' or \

highlight > 100:

raise ValueError("sorry, you lose")

if width == 0 and height == 0 and (color == 'red' or

emphasis is None):

raise ValueError("I don't think so -- values are %s, %s" %

(width, height))

Blob.\_\_init\_\_(self, width, height,

color, emphasis, highlight)

Blank Lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related

functions. Blank lines may be omitted between a bunch of related

one-liners (e.g. a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the control-L (i.e. ^L) form feed character as whitespace;

Many tools treat these characters as page separators, so you may use them

to separate pages of related sections of your file.

Encodings ([PEP 263](http://www.python.org/dev/peps/pep-0263))

Code in the core Python distribution should aways use the ASCII or

Latin-1 encoding (a.k.a. ISO-8859-1). For Python 3.0 and beyond,

UTF-8 is preferred over Latin-1, see [PEP 3120](http://www.python.org/dev/peps/pep-3120).

Files using ASCII (or UTF-8, for Python 3.0) should not have a

coding cookie. Latin-1 (or UTF-8) should only be used when a

comment or docstring needs to mention an author name that requires

Latin-1; otherwise, using \x, \u or \U escapes is the preferred

way to include non-ASCII data in string literals.

For Python 3.0 and beyond, the following policy is prescribed for

the standard library (see [PEP 3131](http://www.python.org/dev/peps/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 MUST provide a latin transliteration of their

names.

Open source projects with a global audience are encouraged to

adopt a similar policy.

**Imports**

- Imports should usually be on separate lines, e.g.:

Yes: import os

import sys

No: import sys, os

it's okay to say this though:

from subprocess import Popen, PIPE

- 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:

1. standard library imports

2. related third party imports

3. local application/library specific imports

You should put a blank line between each group of imports.

Put any relevant \_\_all\_\_ specification after the imports.

- Relative imports for intra-package imports are highly discouraged.

Always use the absolute package path for all imports.

Even now that [PEP 328](http://www.python.org/dev/peps/pep-0328) [7] is fully implemented in Python 2.5,

its style of explicit relative imports is actively discouraged;

absolute imports are more portable and usually more readable.

- When importing a class from a class-containing module, it's usually okay

to spell this

from myclass import MyClass

from foo.bar.yourclass import YourClass

If this spelling causes local name clashes, then spell them

import myclass

import foo.bar.yourclass

and use "myclass.MyClass" and "foo.bar.yourclass.YourClass"

**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

- Immediately before the open parenthesis that starts the argument

list of a function call:

Yes: spam(1)

No: spam (1)

- Immediately before the open parenthesis that starts an indexing or

slicing:

Yes: dict['key'] = list[index]

No: dict ['key'] = list [index]

- More than one space around an assignment (or other) operator to

align it with another.

Yes:

x = 1

y = 2

long\_variable = 3

No:

x = 1

y = 2

long\_variable = 3

Other Recommendations

- Always surround these binary operators with a single space on

either side: assignment (=), augmented assignment (+=, -= etc.),

comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not),

Booleans (and, or, not).

- Use spaces around arithmetic operators:

Yes:

i = i + 1

submitted += 1

x = x \* 2 - 1

hypot2 = x \* x + y \* y

c = (a + b) \* (a - b)

No:

i=i+1

submitted +=1

x = x\*2 - 1

hypot2 = x\*x + y\*y

c = (a+b) \* (a-b)

- Don't use spaces around the '=' sign when used to indicate a

keyword argument or a default parameter value.

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()

Rather not:

if foo == 'blah': do\_blah\_thing()

do\_one(); do\_two(); do\_three()

- While sometimes it's okay to put an if/for/while with a small

body on the same line, never do this for multi-clause

statements. Also avoid folding such long lines!

Rather not:

if foo == 'blah': do\_blah\_thing()

for x in lst: total += x

while t < 10: t = delay()

Definitely not:

if foo == 'blah': do\_blah\_thing()

else: do\_non\_blah\_thing()

try: something()

finally: cleanup()

do\_one(); do\_two(); do\_three(long, argument,

list, like, this)

if foo == 'blah': one(); two(); three()

**Comments**

Comments that contradict the code are worse than no comments. Always make

a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or

sentence, its first word should be capitalized, unless it is an identifier

that begins with a lower case letter (never alter the case of

identifiers!).

If a comment is short, the period at the end can be omitted. Block

comments generally consist of one or more paragraphs built out of complete

sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, Strunk and White apply.

Python coders from non-English speaking countries: please write

your comments in English, unless you are 120% sure that the code

will never be read by people who don't speak your language.

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 #.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline

comments should be separated by at least two spaces from the statement.

They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state

the obvious. Don't do this:

x = x + 1 # Increment x

But sometimes, this is useful:

x = x + 1 # Compensate for border

**Documentation Strings**

Conventions for writing good documentation strings (a.k.a. "docstrings")

are immortalized in [PEP 257](http://www.python.org/dev/peps/pep-0257) [3].

- 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](http://www.python.org/dev/peps/pep-0257) describes good docstring conventions. Note that most

importantly, the """ that ends a multiline docstring should be on a line

by itself, and preferably preceded by a blank line, e.g.:

"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""

- For one liner docstrings, it's okay to keep the closing """ on the same

line.

**Version Bookkeeping**

If you have to have Subversion, CVS, or RCS crud in your source file, do

it as follows.

\_\_version\_\_ = "$Revision: 68852 $"

# $Source$

These lines should be included after the module's docstring, before any

other code, separated by a blank line above and below.

**Naming Conventions**

The naming conventions of Python's library are a bit of a mess, so we'll

never get this completely consistent -- nevertheless, here are the

currently recommended naming standards. New modules and packages

(including third party frameworks) should be written to these standards,

but where an existing library has a different style, internal consistency

is preferred.

Descriptive: Naming Styles

There are a lot of different naming styles. It helps to be able to

recognize what naming style is being used, independently from what they

are used for.

The following naming styles are commonly distinguished:

- b (single lowercase letter)

- B (single uppercase letter)

- lowercase

- lower\_case\_with\_underscores

- UPPERCASE

- UPPER\_CASE\_WITH\_UNDERSCORES

- CapitalizedWords (or CapWords, or CamelCase -- so named because

of the bumpy look of its letters[4]). This is also sometimes known as

StudlyCaps.

Note: When using abbreviations in CapWords, capitalize all the letters

of the abbreviation. Thus HTTPServerError is better than

HttpServerError.

- mixedCase (differs from CapitalizedWords by initial lowercase

character!)

- Capitalized\_Words\_With\_Underscores (ugly!)

There's also the style of using a short unique prefix to group related

names together. This is not used much in Python, but it is mentioned for

completeness. For example, the os.stat() function returns a tuple whose

items traditionally have names like st\_mode, st\_size, st\_mtime and so on.

(This is done to emphasize the correspondence with the fields of the

POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python,

this style is generally deemed unnecessary because attribute and method

names are prefixed with an object, and function names are prefixed with a

module name.

In addition, the following special forms using leading or trailing

underscores are recognized (these can generally be combined with any case

convention):

- \_single\_leading\_underscore: weak "internal use" indicator. E.g. "from M

import \*" does not import objects whose name starts with an underscore.

- single\_trailing\_underscore\_: used by convention to avoid conflicts with

Python keyword, e.g.

Tkinter.Toplevel(master, class\_='ClassName')

- \_\_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.

Prescriptive: Naming Conventions

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.

Package and Module Names

Modules should have short, all-lowercase names. Underscores can be used

in the module name if it improves readability. Python packages should

also have short, all-lowercase names, although the use of underscores is

discouraged.

Since module names are mapped to file names, and some file systems are

case insensitive and truncate long names, it is important that module

names be chosen to be fairly short -- this won't be a problem on Unix,

but it may be a problem when the code is transported to older Mac or

Windows versions, or DOS.

When an extension module written in C or C++ has an accompanying Python

module that provides a higher level (e.g. more object oriented)

interface, the C/C++ module has a leading underscore (e.g. \_socket).

Class Names

Almost without exception, class names use the CapWords convention.

Classes for internal use have a leading underscore in addition.

Exception Names

Because exceptions should be classes, the class naming convention

applies here. However, you should use the suffix "Error" on your

exception names (if the exception actually is an error).

Global Variable Names

(Let's hope that these variables are meant for use inside one module

only.) The conventions are about the same as those for functions.

Modules that are designed for use via "from M import \*" should use the

\_\_all\_\_ mechanism to prevent exporting globals, or use the older

convention of prefixing such globals with an underscore (which you might

want to do to indicate these globals are "module non-public").

Function Names

Function names should be lowercase, with words separated by underscores

as necessary to improve readability.

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 "print\_" is better than

"prnt". (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.

Python mangles these names with the class name: if class Foo has an

attribute named \_\_a, it cannot be accessed by Foo.\_\_a. (An insistent

user could still gain access by calling Foo.\_Foo\_\_a.) Generally, double

leading underscores should be used only to avoid name conflicts with

attributes in classes designed to be subclassed.

Note: there is some controversy about the use of \_\_names (see below).

Constants

Constants are usually declared on a module level and written in all

capital letters with underscores separating words. Examples include

MAX\_OVERFLOW and TOTAL.

Designing for inheritance

Always decide whether a class's methods and instance variables

(collectively: "attributes") should be public or non-public. If in

doubt, choose non-public; it's easier to make it public later than to

make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your

class to use, with your commitment to avoid backward incompatible

changes. Non-public attributes are those that are not intended to be

used by third parties; you make no guarantees that non-public attributes

won't change or even be removed.

We don't use the term "private" here, since no attribute is really

private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the "subclass

API" (often called "protected" in other languages). Some classes are

designed to be inherited from, either to extend or modify aspects of the

class's behavior. When designing such a class, take care to make

explicit decisions about which attributes are public, which are part of

the subclass API, and which are truly only to be used by your base

class.

With this in mind, here are the Pythonic guidelines:

- 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. (However,

notwithstanding this rule, 'cls' is the preferred spelling for any

variable or argument which is known to be a class, especially the

first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

- For simple public data attributes, it is best to expose just the

attribute name, without complicated accessor/mutator methods. Keep in

mind that Python provides an easy path to future enhancement, should

you find that a simple data attribute needs to grow functional

behavior. In that case, use properties to hide functional

implementation behind simple data attribute access syntax.

Note 1: Properties only work on new-style classes.

Note 2: Try to keep the functional behavior side-effect free, although

side-effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive

operations; the attribute notation makes the caller believe

that access is (relatively) cheap.

- 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.

Note 1: Note that only the simple class name is used in the mangled

name, so if a subclass chooses both the same class name and attribute

name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and

\_\_getattr\_\_(), less convenient. However the name mangling algorithm

is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the

need to avoid accidental name clashes with potential use by

advanced callers.

**Programming Recommendations**

- Code should be written in a way that does not disadvantage other

implementations of Python (PyPy, Jython, IronPython, Pyrex, Psyco,

and such).

For example, do not rely on CPython's efficient implementation of

in-place string concatenation for statements in the form a+=b or a=a+b.

Those statements run more slowly in Jython. In performance sensitive

parts of the library, the ''.join() form should be used instead. This

will ensure that concatenation occurs in linear time across various

implementations.

- Comparisons to singletons like None should always be done with

'is' or 'is not', never the equality operators.

Also, beware of writing "if x" when you really mean "if x is not None"

-- e.g. when testing whether a variable or argument that defaults to

None was set to some other value. The other value might have a type

(such as a container) that could be false in a boolean context!

- Use class-based exceptions.

String exceptions in new code are forbidden, because this language

feature is being removed in Python 2.6.

Modules or packages should define their own domain-specific base

exception class, which should be subclassed from the built-in Exception

class. Always include a class docstring. E.g.:

class MessageError(Exception):

"""Base class for errors in the email package."""

Class naming conventions apply here, although you should add the suffix

"Error" to your exception classes, if the exception is an error.

Non-error exceptions need no special suffix.

- When raising an exception, use "raise ValueError('message')" instead of

the older form "raise ValueError, 'message'".

The paren-using form is preferred because when the exception arguments

are long or include string formatting, you don't need to use line

continuation characters thanks to the containing parentheses. The older

form will be removed in Python 3000.

- When catching exceptions, mention specific exceptions

whenever possible instead of using a bare 'except:' clause.

For example, use:

try:

import platform\_specific\_module

except ImportError:

platform\_specific\_module = None

A bare 'except:' clause will catch SystemExit and KeyboardInterrupt

exceptions, making it harder to interrupt a program with Control-C,

and can disguise other problems. If you want to catch all

exceptions that signal program errors, use 'except Exception:'.

A good rule of thumb is to limit use of bare 'except' clauses to two

cases:

1) If the exception handler will be printing out or logging

the traceback; at least the user will be aware that an

error has occurred.

2) If the code needs to do some cleanup work, but then lets

the exception propagate upwards with 'raise'.

'try...finally' is a better way to handle this case.

- Additionally, for all try/except clauses, limit the 'try' clause

to the absolute minimum amount of code necessary. Again, this

avoids masking bugs.

Yes:

try:

value = collection[key]

except KeyError:

return key\_not\_found(key)

else:

return handle\_value(value)

No:

try:

# Too broad!

return handle\_value(collection[key])

except KeyError:

# Will also catch KeyError raised by handle\_value()

return key\_not\_found(key)

- Use string methods instead of the string module.

String methods are always much faster and share the same API with

unicode strings. Override this rule if backward compatibility with

Pythons older than 2.0 is required.

- Use ''.startswith() and ''.endswith() instead of string slicing to check

for prefixes or suffixes.

startswith() and endswith() are cleaner and less error prone. For

example:

Yes: if foo.startswith('bar'):

No: if foo[:3] == 'bar':

The exception is if your code must work with Python 1.5.2 (but let's

hope not!).

- Object type comparisons should always use isinstance() instead

of comparing types directly.

Yes: if isinstance(obj, int):

No: if type(obj) is type(1):

When checking if an object is a string, keep in mind that it might be a

unicode string too! In Python 2.3, str and unicode have a common base

class, basestring, so you can do:

if isinstance(obj, basestring):

In Python 2.2, the types module has the StringTypes type defined for

that purpose, e.g.:

from types import StringTypes

if isinstance(obj, StringTypes):

In Python 2.0 and 2.1, you should do:

from types import StringType, UnicodeType

if isinstance(obj, StringType) or \

isinstance(obj, UnicodeType) :

- For sequences, (strings, lists, tuples), use the fact that empty

sequences are false.

Yes: if not seq:

if seq:

No: if len(seq)

if not len(seq)

- Don't write string literals that rely on significant trailing

whitespace. Such trailing whitespace is visually indistinguishable and

some editors (or more recently, reindent.py) will trim them.

- Don't compare boolean values to True or False using ==

Yes: if greeting:

No: if greeting == True:

Worse: if greeting is True:

**References**

[1] [PEP 7](http://www.python.org/dev/peps/pep-0007), Style Guide for C Code, van Rossum

[2] <http://www.python.org/doc/essays/styleguide.html>

[3] [PEP 257](http://www.python.org/dev/peps/pep-0257), Docstring Conventions, Goodger, van Rossum

[4] <http://www.wikipedia.com/wiki/CamelCase>

[5] Barry's GNU Mailman style guide

<http://barry.warsaw.us/software/STYLEGUIDE.txt>

[6] [PEP 20](http://www.python.org/dev/peps/pep-0020), The Zen of Python

[7] [PEP 328](http://www.python.org/dev/peps/pep-0328), Imports: Multi-Line and Absolute/Relative

**Copyright**

This document has been placed in the public domain.