

PEP 8 -- Style Guide for python code

• Introduction -

This document gives coding conventions for the python code comprising the standard library in the main python distribution. The companion informational PEP describing style guidelines for the C code in the implementation of Python.

PEP8 and PEP257 (Docstring document conventions) were adapted from Guido's original python style guide essay, with some additions from Barry's style guide.

This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

Many projects have their own coding style guidelines. In this event of any conflict, such project-specific guides take precedence for that project.

• 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 & make it consistent across the wide

spirit spectrum of python code As PEP 20 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 the most important.

however, know when to be inconsistent - sometimes style guide recommendations just aren't applicable. when in doubt, use your best judgement. look at other examples and decide what looks best and don't hesitate to ask!

In particular: do not break backwards compatibility just to comply with this PEP!

some other good reasons to ignore a particular guideline:

1. when applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.
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).

- 3 Because the code in question predates the introduction of the guideline and there is no other reason to be modifying that code.
- 4 When the code needs to remain compatible with older versions of python that don't support the feature recommended by the style guide.

Code lay out

Indentation

use 4 spaces per intro indentation level.

continuation lines should align wrapped elements either vertically using python's `\` line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following should be considered: There should be no arguments on the first line and further indentation should be clearly distinguish itself as a continuation line:

correct:

Aligned with opening delimiter

foo =

long_function_name(var_one,
var_two,

var three, var four)

Add a 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 indnts should add a level.

```
foo = long_function_name (
    var one, var two,
    var three, var four)
```

wrong :

Arguments on first line
forbidden when not using
vertical alignment

```
foo =
long_function_name (var one,
var two,
var three, var four).
```

further indentation required
as indentation is not distinguishable

```
def long_function_name (
    var one, var two, var three,
    var four):
    print (var one)
```

The 4-space continuation is optional for continuation lines.

Optional:

Hanging indent *may* be indented to other than 4 spaces.

```
foo = long-function-name (  
    var_one, var_two  
    var_three, var_four ),
```

When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it's worth noting that the combination of a two character keyword (i.e. if), plus a single space, plus an opening parenthesis creates a natural 4-space indent for the subsequent lines of the multiline conditional. This can produce a visual conflict with the intended suite of code nested inside the if-statement, which would also naturally be intended to 4 spaces. This PEP takes no explicit position on how (or whether) to further visually distinguish such conditional lines from the nested suite inside the if-statement. Acceptable options in this situation include, but are not limited to:

NO extra indentation

```
if ( this_is_one_thing and  
    that_is_another_thing ):  
    do_something ()
```

Add a comment, which will provide some distinction in editors

supporting syntax highlighting

if (this is one thing and
that is another thing):

since both conditions are
true, we can grobniate
do something ()

Add some extra indentation on
the conditional continuation
line.

if (this is one thing
and

that is another thing):

do something ()

The closing brace/bracelet) parenthesis on
multiline comments may either line up under
the first non-white space character of the last
line of list, as in

```
my_list = [  
    1, 2, 3  
    4, 5, 6,  
]
```

result =

```
some function that takes arguments(  
    'a', 'b', 'c',  
    'd', 'e', 'f',  
)
```

or it may be lined up under the first character of the line that starts the multiline construct as in:

```
my_list = [
```

```
    1, 2, 3,
```

```
    4, 5, 6
```

```
]
```

```
result =
```

some function that takes arguments (

```
'a', 'b', 'c',
```

```
'd', 'e', 'f',
```

```
)
```

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 disallows mixing tabs and spaces for indentation

Maximum line length

limit all lines to a maximum of 79 characters

for flowing long blocks of text with few structural restrictions (delimiters or comments), the line length should be limited to 72 characters

limiting the required editor window width makes it possible to have several files open side by side, and work well when using code review tools that present the two versions in adjacent columns.

The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chosen to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic line wrapping at all.

Some teams strongly prefer a longer line length for code maintained exclusively or primarily by a team that can reach agreement on this issue; it is okay to increase the line length limit up to 99 characters, provided that comments and docstrings are still wrapped at 72 characters.

The Python standard library is conservative and requires limited lines to 79 characters (and docstrings/comments to 72).

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by

wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation.

Backslashes may still be appropriate at times, for example, long, multiple with statements could not use implicit continuation before python 3.10, so backslashes were acceptable for that case :

with

```
open ('/path/to/some/file/you/want/to/  
read') as file_1, \
```

```
open ('/path/to/some/file-being/written', 'w')  
as file_2:
```

```
file_2.write(file_1.read())
```

(see the previous discussion on multiline if-statement for further thoughts on the indentation of such multiline with statements.)

Another such case is with assert statements

make sure to indent the continued line appropriately.

Line break before or after a binary operator

For decades the recommended style was to break after binary operators. But this can hurt readability in two ways: the operators tend to get scattered across different columns on the screen, and each operator is moved away from its operand and onto the previous line. Here, the eye has to do extra work to tell which items are added and which are subtracted.

wrong:

operators sit far away from their operands

$$\text{income} = (\text{gross-wages} + \text{taxable-interest} + (\text{dividends} - \text{qualified-dividends}) - \text{ira-deduction} - \text{student-loan-interest})$$

To solve this readability problem, mathematicians and their publishers follow the opposite convention. Donald Knuth explains the traditional rule in his *Computers and Typesetting* series: "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations."

following the tradition from mathematics usually results in more readable code!

correct:

easy to match operators with operands

income = (gross_wages

+ taxable_interest

+ (dividends -

qualified_dividends)

- ira_deduction

-

student_loan_interest)

In python code, it is permissible to break before or after a binary operator as long as the conversation is consistent locally. For new code Knuth's style is suggested.

Blank lines

Surround top-level function and class definition with two blank lines.

Method definitions inside a class are surrounded 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 (`\f`) form feed character as whitespace; many tools treat these characters as page separators, so may use them to separate pages of related sections of your file note, some editors and web-based code viewers may not recognize control-L as a form feed and will show and other glyph in its place.

Source File Encoding

code in the core python distribution should always use UTF-8, and should not have an encoding declaration.

In the standard library, non-UTF-8 encodings should be used only for test purposes or use non-ASCII characters sparingly, preferably only to denote place and human names. If using non ASCII characters as data, avoid noisy uniced characters like zalgo and byte order marks.

All identifiers in the python standard library must use ASCII-only identifiers, and should use English words whenever feasible (in many cases, abbreviations and technical terms are used which aren't English.)

Open source projects with a global audience are encouraged to adopt a similar policy.

Imports

- Imports should usually be on separate lines :

correct :

```
import os  
import sys
```

wrong :

```
import sys, os
```

It's okay to say this though :

correct :

```
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.
- * A blank line is must between each group of imports.

- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path).

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import
example
```

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import sibling
from sibling import example.
```

Standard library code should avoid complex package layouts and always use absolute imports.

- When importing a class from a class-module containing module, it's usually okay to spell this:

```
from myclass import MyClass
from foo.bar import MyClass import
YourClass
```

If this spelling causes local name clashes,
then spell them explicitly:

```
import MyClass  
import foo.bar.yourClass
```

and use "MyClass MyClass" and
"foo.bar.yourClass yourClass" -

- Wildcard imports (from <module> import *)
should be avoided, as they make it unclear
which names are present in the namespace,
confusing both readers and many automated
tools. There is one defensible use case for a
wildcard import, which is to republish an
internal interface as part of a public API (for
example, overwriting a pure Python implementation
of an interface as part of a with the definition
from an optional accumulator module and
exactly which definitions will be overwritten
isn't known in advance).

When republishing names this way, the guidelines
below regarding public and internal interfaces
still apply.

Module level dunder names

module level "dunders" (ie names with two leading and two trailing underscores) such as `__all__`, `__author__`, `__version__`, etc., should be placed after the module docstring but before any import statements except from `future imports`. Python mandates that `future-imports` must appear in the module before any other code except docstrings:

```
""" This is the example module.
```

```
This module is stuff
```

```
from __future__ import  
barry_as_FLUFL
```

```
-- __all__ = ['a', 'b', 'c']
```

```
-- __version__ = '0.1'
```

```
-- __author__ = 'cardinal biggus'
```

```
import os  
import sys.
```

String quotes

In python, single quoted strings and double-quoted strings are the same. This PEP does not make a recommendation for this pick a rule and stick to it. When a string contains single or double quote characters, however, use the other one to avoid backslashes in the string. It improves readability.

for triple-quoted strings, always use double quote characters to be consistent with the docstring convention in PEP257

Whitespace in Expressions & statements

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

correct:

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

wrong:

```
spam(ham [1], 1 eggs:2)
```

- Between a trailing comma and following close parenthesis:

correct:

foo = (0,)

wrong:

bar = (0,)

- Immediately before a comma, semicolon, or colon:

correct:

if x == 4: print(x, y); x, y = y, x

wrong:

if x == 4: print(x, y); x,
y = y, x

- However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted:

correct:

ham[1:9], ham[1:9:3],
ham[:9:3], ham[1::3],
ham[1:9:]

ham [lower:upper],
ham [lower:upper],
ham [lower :: step]
ham [lower + offset :
upper + offset]
ham [: upper_fn(x) :
step_fn(x)], ham[: :
step_fn(x)]
ham [lower + offset : upper +
offset]

wrong

ham[lower + offset:upper +
offset]ham[1:9], ham[1:9], ham[1:9
: 3]

ham[lower : :upper]

ham [:upper]

- Immediately before the open parenthesis that starts the argument list of a function call

correct:

spam(1)

wrong:

spam(1).

- Immediately before the open parenthesis that starts an indexing or slicing:

correct:

```
dict['key'] = list[index]
```

wrong:

```
dict ['key'] = list [index]
```

- more than one space around an assignment (or other) operator to align it with another:

correct:

```
x = 1
```

```
y = 2
```

```
long-variable = 3
```

wrong:

```
x = 1
```

```
y = 2
```

```
long-variable = 3
```

- Avoid trailing whitespace anywhere because it's usually invisible, it can be confusing:
e.g. a backslash followed by space and a newline does not count as a line continuation marker. Some editors don't preserve it and many projects (like Python itself) have pre-commit hooks that reject it.

- Always surround these binary operators with a single space on either side: assignment (`=`), augmented assignment (`+=`, `-=` etc.), comparison (`==`, `<`, `>`, `!=`, `<=`, `>=`, `in`, `not in`, `is`, `is not`) boolean (`and`, `or`, `not`).
- If operators with different priorities are used consider adding whitespace around the operator with the lowest priority(es). Use your own judgement, however, never use more than one space, and always have the same amount of whitespace on both sides of binary operator.

correct:

 $i = 1 + 1$ $\text{submitted } += 1$ $x = x^* 2 - 1$ $\text{hypot2} = x^* x + y^* y$ $c = (a+b) * (a-b)$

wrong:

 $i = i + 1$ $\text{submitted } += 1$ $x = x^* 2 - 1$ $\text{hypot2} = x^* x + y^* y$ $c = (a+b) * (a-b)$

- function annotations should use the normal rules for colons and always have spaces around the `->` arrow if present.

Function annotations:

correct:

```
def munge (input: AnyStr): ...  
def munge () -> PostInt: ...
```

wrong:

```
def munge (input: AnyStr): ...  
def munge () -> PostInt: ...
```

- 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.

correct:

```
def complex (real, imag= 0.0):  
    return magic (r= real,  
                  i= imag)
```

wrong:

```
def complex (real, imag = 0.0):  
    return magic (r = real, i =  
                  imag)
```

When combining an argument annotation with a default value, however, do use spaces around the = sign!

correct:

```
def munge (sep: AnyStr = None):  
    ...  
def munge (input: AnyStr, sep:
```

Anystr=None, limit=1000):

wrong:

```
def munge(input: Anystr=None):
```

...
def munge(input: Anystr, limit
= 1000): ...

- compound statements (multiple statements on the same line) are generally discouraged.

correct:

```
if foo == 'blah':
```

```
    do_bla...h-thing()  
    do_one()  
    do_two()  
    do_three()
```

Rather not:

wrong:

```
if foo == 'blah':
```

```
    do_bla...h-thing()
```

```
    do_one(); do_two(); do_three()
```

- while sometime 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 :

wrong

if foo == 'blah':

do_blaah_thing()

for x in list: total += x

while t < 10: t = delay()

Definitely not :

wrong:

if foo == 'blah':

do_blaah_thing()

else: do_non_blaah_thing()

try: something()

finally: cleanup()

do_one(); do_two();

do_three(long_argument,

list, like, this)

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

two(); three()

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).

Designing for inheritance

always decide whether a class methods and instance variables (collectively "attributes") should be public or non-public. If in doubt, choose non-variable 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 backwards 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 them 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 behaviour. 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 base class.