

PEP 8 -- style Guide for Python Code.

Introduction:

The PEP is an abbreviation from of python Enterprise proposal. writing code with proper logic is a key factor of programming, but many other important factors can affect the code's quality. The developer's coding style makes the code much reliable, and every developer should keep in mind that python strictly follows the way of order and format of the string.

PEP is a document that provides various guidelines to write the readable in python. PEP 8 describes how the developer can write beautiful code. It was officially written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan. The main aim of PEP is to enhance the readability and consistency of code.

* Code Lay-out

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. When using a hanging the following should be considered; there should be

no arguments on the first line and further itself as a continuation line.

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)
```

The 4 - space rule is optional for continuation lines.

Hanging indents * 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

plus an opening parenthesis creates a natural 4-space indent for the sub-sequent lines of the multiline conditional. This can produce a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented the 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:

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 fabricate.
do-something()

Add some extra indentation on the conditional continuation line.

if (this - is - one - thing)
and that - is - another - thing):
do something()

(Also see the discussion of whether to break before or after binary operators below.)

* 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 following long blocks of text with fewer structural restrictions (docstrings 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 works 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.

Backslashes may still be appropriate at times. For examples, long, multiple with - statements could not use implicit continuation before Python 3.10, so backslashes were acceptable for that cases.

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())

* Should a 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 operations tend to get scattered across different columns on the screen, and each operator is moved away from its operand and on to the previous lines. 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
income = (gross - wages +
taxable - wages +
(dividend - 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 convention is consistent locally, for new code Knuth's style is suggested.

* Blank Lines.

Surround top-level function and class definitions 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 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. Note, some editors and web-based code viewers may not recognize control-L as form feed and will show another 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.

All identifiers in the python standard and library MUST use ASCII-only identifiers, and should use non-ASCII characters, sparingly, preferably only to denote places and human names. If using non-ASCII characters as data, avoid noisy unicode characters like zalgo and byte order marks.

All identifiers in 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).

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

* Imports.

Imports should usually be on separate lines:

correct:

```
import os
```

```
import sys
```

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.
- you should put a blank line 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 error 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, explicitly 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 containing module, it's usually okay to spell this:

```
from myclass import Myclass from foo.bar.  
yourclass import your class.
```

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.Your class".
```

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 with the definitions from an optional accelerator 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" (i.e. names with two leading and two trailing underscores) such as `__all__`, `__author__`, `__version__` etc. should be placed after the module document 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 is module does stuff.
    """
```

```
from __future__ import
    barry as FLUFL
__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'
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 PEP 257.

* Whitespace in Expressions and Statements.

Pet Peeves

Avoid extraneous whitespace in the following situations:
Immediately inside parentheses, brackets or braces:

correct:

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

Between a trailing comma and a following close parenthesis:

correct:

```
foo = (0),
```

Immediately before a comma, semicolon, or colon.

correct:

```
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 items must have the same amount of spacing applied. Exceptions: 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::step]

ham [lower + offset : upper + offset]

ham [: upper - fn(x) : step - fn(x)],

ham [:! step - fn(x)]

ham [lower + offset : upper + offset]

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

correct:

dict ['key'] = 1st [index]

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

correct:

x = 1

y = 2

long. variable = 3

* Other Recommendations

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

correct:

i = i + 1

submitted += 1

x = x * 2 - 1

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. (See function Annotations below for more about function annotations.):

correct:

def munge (input: Anystr): ...

```
def munge() → PosInt ....
```

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 . image = 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): ...
```

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

correct:

```
if foo == 'blah':  
    do - blah - thing()
```

```
do one - ()
```

```
do - two ()
```

```
do - three ()
```


while sometimes it's okay to put an `if/for/while` a small body on the same line, never do this for multiclause statements. Also avoid flooding such long lines!

* When to Use Trailing Commas

Trailing commas are usually optional, except they are mandatory when making a tuple of one element. For clarity, it is recommended to surround the latter in (technically redundant) parentheses:

correct:

```
FILES : ('setup.cfg')
```

wrong

```
FILES = 'setup.cfg'
```

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

* 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!

Ensures that your comments are clear and easily understandable to other speakers of the language you are writing in.

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.

Write docstrings for all public modules, functions, classes and methods. Docstrings are not necessary for non-public method.

""" Return a foobag

Optional plotz says to frobnicate the
bizbaz first.
"""

For one liner docstrings, please keep the
closing """ on the same line:

""" Return an ex-parrot. """

* 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.

* Overriding Principle

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