Fyit-58 Rishi Pant.

PEP 8 -- Python

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

This document & PEP 257 (Docstring conventions) were adapted from Guido's original python style Guide essay, with some additions from Barry's style guide.

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

Many projects have their own coding style guidelines. In the event of any conflicts, such project-specific guides takes 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 spectrum of python code. As PEP 20 says, "Readability counts".

A style guide is about 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 judgment. Look at other example & decide what looks best. And don't hesitate to ask!

In particular: do not break backwards compability just to comply 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 there 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 & 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 indentation level.

Continuation lines should align wrapped elements either vertically using python's implicit line joining inside parentheses, brackets & braces, or using a hanging indent [1]. 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:

Correct:-

# 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)
```

wrong:#

Arguments on first 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 rule is optional for continuation lines

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 produces a visual conflict with the indented suite of code nested inside the if-statement, which would also naturally be indented to 4 spaces. This PEP takes no explicit position on how 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
  frobnicate.
do_something()
```

Add some extra indentation on the continuation line.

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

The closing brace/bracket/parenthesis on multi-line constructs may either line up under the first non-whitespace 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 multi-line construct, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_argument
s(
    '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 & spaces for indentation.

Maximum line length

Limit all lines to a maximum of 79 characters.

For flowing long blocks of text with fewer structions (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, & works well when using code review tool 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 & docstrings are still wrapped at 72 characters.

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

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets & 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())
```

Another such case is with assert statements.

Make sure to indent the continued line appropriately.

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 operators tend to get scattered across different columns on the screen, & each operator is moved away from its operand & onto the previous line. Here, the eye has to do extra work to tell which items are added & which are subtracted:

wrong:
operators sit far away from their operands.

```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

To solve this readability problem, mathematicians & their publishers follow the opposite convention. Donald Knuth explains the traditional rule in his Computers & Typesetting series: "Although formulas within a paragraph always break after binary operations & 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 convention is consistent locally. For new code Knuth's style is suggested.

Blank Lines

Surround top-level function & 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-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. Note, some editors & web-based code viewers may not recognize control-L as a form feed & will show another glyph in its place.

Source file Encoding

Code in the core python distribution should always use UTF-8, & should not an encoding declaration.

In the standard library, non-UTF-8 encodings should be used only for test purposes. Use non-ASCII characters sparingly, preferable only to denote places & human names. If using non-ASCII characters as data, avoid noisy unicode characters like Zalgo & byte order marks.

All identifiers in the python standard library MUST use ASCII-only identifiers, & should use English words wherever feasible (in many cases, abbreviations & 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 & docstrings, & before module globals & 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, 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 & 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 Yourclass.

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

import myclass
import foo.bar.yourclass.

and use "myclass.Myclass" & "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 & 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 accelarator module & exactly.

which definitions will be over-written in advance).

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

Module Level Dunder Names

Module level "dunders" (i.e. names with two leading & two trailing underscores) such as `__all__`, `__author__`, `__version__`, etc. should be placed after the module docstring but before any import statement 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 does stuff.
"""

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'
```

Page No.: 18
Date: / /

```
import os
import sys
```

String quotes

In python, single-quoted strings & double-quoted strings are the same. this PEP does not make a recommendation for this, pick a rule & 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.

White space in expressions & statements

Pet peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

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

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

- Between a trailing comma & a 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, & 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:

dct['key'] = lst[index]

wrong:

dct ['key'] = lst [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.
```