F4IT_42_Rohit

IT TOOLS ASSIGNMENT

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. Please see the companion informationed PEP describing style guidelines for the C code in the C implementation of Python.

This document and PEP 257 (Docstring Conventions) were adopted from Guido's original Python Style Guide essay, with some additions from Barry's style guide[2].

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

Many projects have their own coding style guidelines. In the event of any conflicts, 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 and make it consistent across the wide 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.

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 :

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 ~~a~~ h indent [I]. When using

Page No.
Date

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 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 rule is optional for continuation lines.

Optional:

```
# 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 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 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 (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 distraction in editors

# Supporting syntax highlighting.
if (this_is_one_thing and
    that_is_another_thing):
```

# Since both conditions are true, we can consolidate.
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.)

The closing brace/bracket/parenthesis on multiline 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 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 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. 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 upto 99 characters, provided that comments and docstrings are still wrapped at 72 characters.

The Python standard library is conservative and requires limiting line to 79 characters (and docstrings / comments and docstring / 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 - statement could not use implicit continuation before Python 3.10.0, 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-statements 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.

◇ 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, and each operator is moved away from its operand and onto

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 income = (gross_wages +
                   taxable_interest +
                   (dividends - qualified_
                   dividends) - ira_deduct -
                   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" [3].

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 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 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 and web-based code viewers may not recognize control-L as a 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.

In the standard library, non-UTF-8 should be used only for test purposes. 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 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.

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
```

Page No.
Date

- 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, 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-containing module, it's usually okay to spell this:

```
from myclass import MyClass
from foo.bar.yourclass import 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 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 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'

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 contain single or double quote characters, however, use the other one to avoid back-slashes 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.