Pandey Madhusudan Ramkrishpal
FYIT
57

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 informational PEP describing style guideline for the C code in the C implementation of Python.

This document and PEP 257 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 the event of any conflicts, such project-specific guides take precedence for that project.

Page No. 2
Date

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. 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 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!

Roll NO: 57

Some other good reasons to ignore a particular guideline:

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

(ii) 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).

(iii) Because the code in question predates the introduction of the guidence and there is no other reason to be modifying that code.

(iv) When the code needs to remain compatible with older version of Python that don't support the feature recommended by the style guide.

Code Lay-out

Indentation

Use 4 space per indentation level.

Roll No. 57

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 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)
```

Roll NO. 57

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:

Roll NO. 57

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 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:

Roll No. 57

```
# 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
    #  fraboznicate
        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.)

Roll NO. 57

This 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 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 limiting lines to 79 characters.

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. Those should be used in preference to using

Roll NO-57

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, and each operator is moved away from its operand and onto the previous line.

Roll No. 57

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_deduction -
          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 breaks after binary operations and relations, displayed formulas always break before binary operations".

Roll No. 57

Following the tradition from mathematics usually result 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.

Roll No. 57

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 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.

Roll No. 57

In the standard library, non-UTF-8 encodings 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.

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
```

Roll No-57

It's okey 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 if the import system is incorrectly configured:

(such as when a directory inside a package ends up on sys.path)

Roll NO. 57

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

Roll No. 57

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 import (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 over written isn't known in advance).

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

Page No. 19
Date / /

Roll No. 57

Module Level Dunder Names

Module level "dunders" 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 does stuff.
"""

from __future__ import barry_as_FLUFL

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

__version__ = '0.1'

__author__ = 'Cardinal Biggles'

import os

import sys
```

Roll NO. 57

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 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 PEP257.

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})
```

Roll No-57

Wrong

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

- Between a tailing comma and 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, and should have equal

Roll No. 57

amounts on either side. 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]

Roll No. 57

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]

Roll No. 57

- 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
```

Other Recommendations

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: eg. a backslash following by a space and a newline does not count as line continuation marker. Some editors don't preserve it and many projects have pre-commit hooks that reject it.

Roll No-57

- 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).

- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgement; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator:

Correct:

```
i = i + 1

submitted += 1

x = x*2 - 1

hypot2 = x*x + y*y

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

Roll NO. 57

Wrong:

```
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.

Correct:

```
def munge(input: AnyStr): ...

def munge() -> PosInt: ...
```

Wrong:

```
def munge(input:AnyStr): ...

def munge()-> PosInt: ...
```

Roll No-57

- 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): ....
```

Roll No. 57

Wrong:

def munge(input: AnyStr=None): ...

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

- Compound statements are generally discouraged:

correct:

if foo == 'blah':

    do_blah_thing()

do_one()

do_two()

do_three()

Rather not:

wrong:

if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()

Roll No. 57

- While sometimes it's okey 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_blah_thing()

for x in lst: total += x

while t < 10: t = delay()
```

Definitely not:

Wrong:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()
```

Roll No. 57

do_one(); do_two(); do_three(long, argument, list, like, this)

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

⟹X⟸