Name :- Jeet mohan chudasama
Roll no :- 13 class :- FYIT

Page No. 1
Date.

* IT tools *

* Pep 8 *

* 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 and PEP 257 are adapted from Guido's original python style Guide essay, with some additions from Barry's style guide.

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

May projects have their own coding style guide lines. In the event of any conflicts, such project specific guides take precedence for that project.

Scanned with CamScanner

Test chudasama
.13 F7I7

* cd 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. cds PEP 20 says, "Readability counts".

cd 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. cnd don't hesitate to ask!

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

Scanned with CamScanner

Jat chudavana
13 FYIT

Page No. 3
Date

Some other good reasons to ignore a particular guidline :—

1. When applying the guidline 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 ---- although this is also an opportunity to clean up someone else's mess.

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.

Scanned with CamScanner

Jeet chudasama
13 FYIT

Page No. 4
Date

* 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 indent the following should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation:

```
# 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,
```

Scanned with CamScanner

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

Scanned with CamScanner

```
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 nothing that the combination of a two character keyword, plus a single space, plus an opening

Scanned with CamScanner

parenthesis creates a natural 4-space indent for the subsequent lines of the multiline conditional. This can produce a visual effect 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 we 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.
```

Scanned with CamScanner

Jaat chudasama
13. FYIT

Page No. 8
Date.

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()
```

Scanned with CamScanner

Jeet chudasama
13 FYIT

Page No. 9
Date

- The closing brace / brakat / 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',
    )
```

Scanned with CamScanner

- 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',
)
```

Scanned with CamScanner

Jeet chudasama
13 FYIT

Page No. 11
Date

* 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, 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.

Scanned with CamScanner

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 ~~wrap~~ wrapping in editors with the window width set to 80, even ~~if~~ if the tool places a marker ~~glyph~~ 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 docstring are still wrapped at 72 characters.

The Python standard library is conservative and requires limiting lines to 79 characters and docstring / comments to 72.

Scanned with CamScanner

The preffered way of wrapping long lines is by using pythons 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())
```

Scanned with CamScanner

(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 operator. 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:

Scanned with CamScanner

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

Scanned with CamScanner

following the tradition from mathematicians ~~and their publishers follows the opposite convention. Donald~~ 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

Scanned with CamScanner

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-lines eg a set of dummy implementations.

Use blank lines in functions, sparingly to indicate logical sections.

Python accepts the control-L form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related

Scanned with CamScanner

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 encodings should be used only for test purposes 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).

Scanned with CamScanner

Test chudasama
13 FYIT

Page No. 19
Date

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

Its okay to say this though:

correct: ~~to be~~

from subprocess import Popen, PIPE .

Scanned with CamScanner

Jat chudasama
13 FYIT

Page No: 20
Date:

- Imports are always put at the top of the file, just after 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~~ tend to be better behaved or at least give better error messages if the import system is incorrectly

Scanned with CamScanner

* Configured such as when a directly 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..

Scanned with CamScanner