PEP8 - Style Guide for python

Introduction.

This document gives coding convention for the python code comprising the standard library in the main python distribution. Please see the companion Information PEP describing style guidlines for the C code in the C implementation of python. This document and PEP257 (Docstring Convention) were adapted from the Guido's Original python Style Guide essay, with some addition from Barry's style guide. This style guide evolves over time as additional convention are identified and past convention are rendered obsolete by change in the language itself. Many project have their own coding style guideline. In the event of any conflicts, such project-specific guide take precedence for that project.

Code layout

Indentation.

Use 4 spaces per indentation level.

Continuation line should align wrapped elements either vertically using python's implicit line joining inside parentheses, bracket 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. When the conditional part of an if-statment is long enough to require that it be written across multiple line, its worth nothing that the combination of a two character keyword, plus a single space plus an opening parathesis creates a indented suits of code nested inside the if statments, which would also naturally be indented to 4spaces. This can produce a visal conflict with the indent suite of code nested inside the if-statment, which would also naturally be indented to 4space. This PEP takes no explicit position on how to farther visually distinguish such conditional lines from the nested suite inside the if statment.

Acceptable option in this situation include but are not limited to: The closing brace | bracket, | Parantheses on multiline construct my either line up under the first non-whitespace character of the last line of list, as in: or it may be lined up under the first non-whitespace character of the last line as in that starts the multipline construct, as in:

Tabs or Space?

Space 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 restriction, the line length should be limited to 72 characte Limiting the required editor window width makes it possible to have several files open side by side, and work well when using code review tools that represent the two version in adjacent columns. The default wrapping in most tools disrupts th visual structure of the code, making it mo difficult to understand. The limits are c to avoid wrapping in editor with the wi width set to 80, even if the tool place a mark glyph in the final column when wrapping Some web based tools ~~many~~ place wen if the tool places a marker glyph in the fi column when wrapping line at all. Some teams strongly prefer a longer le length. For code maintained exclusive or primarily by a team that can reach agreement on this issue, it is okay to

increase the 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, provide 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 line can be broken over multiple line by wrapping expression in parentheses. These should be used preference to using a backslash for line continuation. Blackslashes may still be appropriate at time. For example, long, multiple with statment could not use implicit continuation before Python 3.10, so backslaches were acceptable for the case.

```
with open('/path/to/some/file/you/want/to/read') as file_1:
    with open('/path/to/some/file/being/written', 'w') as file_2:
        file_2.write(file_1.read())
```

Page No.
Date

(5)

Should a line break before a binary Operator

For decades the recommended style was to break after a binary Operator. But this can hurt readability in two ways: the operators tend to get scattered across different column 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 is subtracted:

```
# Wrong:
# operator sit far away from their
operator
income = (gross_wages +
          taxable_interest +
          (dividends_qualified - dividends) -
          ira_deduction -
          student_loan_interest)
```

To solve this readability problem, mathematicians and their publisher follow the opposite convention. Donal Knuth explains the traditional rule in his Computer & Typesetting series: "Although formulas within a paragraph always break after binary operation and relation, displayed formulas always break before binary operations".

# Correct:
# easy to match operators with operands
income = (gross - wages
+ taxable_interest
+ (dividends - qualified_dividends)
- ira_deduction
- student_loan_interest)

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 line 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.

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 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

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

from . import sibling
from . sibling import example

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 spelling causes local name clashes then spell them explicitly.

import myclass
import foobar.yourclass

Module level Dunder Names

""" 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

Whitespace In Expression and Statments Pet Peeves

- Immediately inside parentheses, bracket or braces:

# Correct:
Spam(ham(1), (eggs:2))

# Wrong:
Spam(ham[1], {eggs:2})

- Between a trailing comma and a following close parenthesis

# Correct:
foo = (0,)

# Wrong:
bar = (0,)

- Immediately before a comma, semicolon, or col

Correct:
if x==4: print(x,4); x,4=4,x

Wrong:

if x ==4 : print (x,4); x,4=4,x

- Immediately before the open parentheses that start the arguments 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]
```

- Name to avoid.

Never use the character 'l' as a single character variable name. In some fonts, these character are indistinguishable from the numerals one and zero, when tempted to use 'l', use 'L' instead.

Constant.

Constants are usually defined on a module level and written in all capital letter with underscores separating word.