PEP8 - Style guide for python code

PEP - 8
Title - Style guide for Python code
Author - Guido van Rossum <guido at python.org>,
Barry Warsaw <barry at python.org>,
Nick Coghlan <ncoghlan at gmail.com>
Status - Active
Type - Process
Created - 05-Jul-2001
Post-History - 05-Jul-2001, 01-Aug-2013

★ Introduction

This document gives coding conventions ~~for~~ 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 style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.

★ 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 consistent - 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!

★ 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. 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_functions_name(
    var_one, var_two, var_three,
    var_four):
```

- 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.

- 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. To solve this readability problem, mathematics 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 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)
```

- Blank Lines

Surround top-level function and class definitions with two blank lines.

★ String Quotes

In python, single quoted strings and double-quoted strings are the same. This PEP does not make a recommendations 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 backslashes in the string. It improves readability.

For triple-quoted string, always use double quoted characters to be consistent with the docstring convention in PEP 257.

★ 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})
```

=> Between a trailing comma and a following close parenthesis:

```
# Correct:
foo = (0,)
```

=> Immediately before a comma, semicolon, or colon:

```
# Correct
if x == 4: print (x, y); x, y = y, x
```

=> Immediately before the open parenthesis that starts the argument list of a function call:

```
# Correct:
Spam (1)
```

=> More than one space around an assignment (or other) operator to align it with another:

```
# Correct
x = 1
y = 2
long-variable = 3
```

=> Immediately before the open parenthesis that starts an indexing or slicing:

```
# Correct:
dct['key'] = lst[index]
```

★ When to use Trailing Commas

Trailing commas are usually optional, except they are mandatory when making a tuple of one element. For clarity, it is recommended to surround the latter in (technically redundant) parentheses:

```
# Correct:
FILES = ('setup.cfg',)
```

When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments or imported items is expected to be extended over time. The pattern is to put each value (etc) on a line by itself, always adding a trailing comma, and add the close parenthesis/bracket/braces on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter (except in the above case of singleton tuples):

```
# correct :
FILES = [
        'setup.cfg',
        'tox.ini',
        ]
initialize (FILES,
            error=True,
            )
```

★ Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!)

• Block Comments

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment) Paragraphs inside a block comment are seperated by a line containing a single #.

Page No.: 7
Date:

- Inline Comments

Use inline comments sparingly.
An inline comment is a comment on the same line as a statement. Inline comments should be seperated by at least two spaces from the statement. They should start with a # and a single space.
Inline comments are unnecessary and in fact distracting if they state the obvious.
Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border.
```

★ Naming Conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent - nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

- Overriding Principle

Names that are visible to the user as public parts of the API should follow conventions that reflect usage

rather than implementation.

- Descriptive : Naming Styles.

There are alot of different naming styles to be able to recognize what naming style is being used, independently from what they are used for.

- Prescriptive : Naming conventions

Names to Avoid:-
Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single variable names.
In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead

ASCII compatibility:-
Identifiers used in the standard library must be ASCII compatible as described in the policy section of PEP 3131.

Package and Module Names:-
Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python package should also have short,

all-lowercase names, although the use of underscored is discouraged.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. _socket).