IT-TOOLS

Siddarth Walkar FYIT 95

PEP 8

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution.
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.

Code Lay-out

~~Tde~~ 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 ~~inde~~ 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.

The 4-space rule is optional for continuation lines.

Option1:
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 space, 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 to further visually distinguish such conditional lines from the nested suit in-side the if statement. Acceptable options in this situation include, but are not limited to.

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:
or it may be lined up under the first character of the line that starts the multiline construct, as in

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 restriction (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 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 (and docstrings/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 multiples 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.

Another such case is with assert statements

Make sure to ~~ident~~ indent the continued line appropriately.

Should a Line Break Before or After a Binary Operator?

For decades the recommended styles way 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. Here the eye has to do extra work to tell which items are added and which are subtracted.

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".

Following the tradition from mathematics usually results in more readable code

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 to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (dummy implementations).

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

Python accepts the control-L (ie ^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 encoding 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 project with a global audience are encouraged to adopt a similar policy

Imports

Imports should usually be on separate lines

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 atleast give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):

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:

Standard & library code should avoid complex package layouts and always use absolute imports.

When importing a class from a class-containing module, its usually okay to spell this:

If this spelling causes local name clashes then spell them explicitly:

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 internal 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~~ 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 ~~doc~~ 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.

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 backslashes in the string. It improves readability.

For triple-quoted strings, always use double quote characters to be consistent with the docstring convention.

Whitespace in Expressions and Statements

Pet Peeves

Avoid extraneous whitespace in the following situations:

Immediately inside parentheses, brackets or braces:

Between a trailing comma and a following close parenthesis:

Immediately before a comma, semicolon or colon:

However in a slice the colon acts like a binary operators and shold have equal 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:

Immediately before the open parenthesis that start the argument list of a function call:

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

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

Other Recommendations

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

Avoid surrounded 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 lowest priority. Use your own judgment; however, never use more than one space and always have the same amount of whitespace on both sides of a binary operator:

Function annotations should use the normal rules for colons and always have spaces around the -> arrow if present.

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:

~~parameter: parameter:~~

When combining an argument annotation with a default value, however, do use spaces around the = sign:

Compound statements (multiple statements on the same line) are generally discouraged

Rather not:

Definitely not:

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 parentheses:

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 on the line by itself, alway adding a trailing comma, and add the close parenthesis / bracket / brace 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):

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 sentence. The first word should be ~~capt~~ capitalized, unless it is an identifier that begins with a lower case letter.

Block comment generally consist of one or more parcegraphs built out of complete sentences, with each sentence ending in a period.

You should use two spaces after a sentence-ending period in multi sentence comments, ~~exp~~ except after the final sentence

Ensure that your comments are clear and easily understable to

other speakers of the language you are writing in.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block Comments

Block comments generally apply to some 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 ~~idented~~ indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly

An inline comments is a comment on the same line as a statement Inline comments should be separated 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:

But sometimes, this is useful:

Documentation Strings

Conventions for writing good documentation string

Write docstring for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods

but you should have a comment that describes what the method does. This comment should appear after the def line

PEP 257 describes good docstring conventions. Note that most importantly the """ that ends a multiline docstring should be on a line by itself:

For one liner docstrings, please keep the closing """ on the same line:

Naming Conventions

The naming convention of Python's library are 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 reflects usage rather than implementation.

Descriptive : Naming Styles

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

The following naming styles are commonly distinguished:

b (single lowercase letter)

B (single uppercase letter)

lowercase

lower_case_with_underscores

UPPERCASE

UPPER_CASE_WITH_UNDERSCORES

Capitalized Words (or CapWords, or CamelCase - so named because of the bumpy look of its letters.
This is also sometimes known as StudlyCaps.

Note: When using acronyms in CapWords, capitalize all letters of the acronyms. Thus HTTPServerError is better than HttpServerError.

mixedCase (differs from CapitalizedWords by initial lowercase character!)

Capitalized_Words_with_Underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example the os.stat() function returns a tuples whose items traditionally have names like st_mode, st_size, st_mtime and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python this style is generally deemed unnecessary because attribute and method names are prefixed with an object and function names are prefixed with a module name.

In addition the following special forms using leading or trailing underscores are recognized

_single_leading_underscore: weak "internal use" indicator. E.g. from M import * does not import objects whose names start with an underscore

single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, e.g.

__double_leading_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo; see below).

__double_leading_and_trailing_underscore__: "magic" objects or attributes that live in user-controlled namespaces. E.g. __init__, __import__ or __file__. Never invent such names; only use them as documented.

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 character 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 ~~se~~ 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 under-scores is discouraged