FyIT-22-Dhananjay

IT-Tools PEP8 ASSIGNMENT.

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 PEP257 (Docstring Conventions) were adapted from Guido's original python style Guide essay, with some additions from Barry's style guide.
This style guide ~~ev~~ evolves over time as additional conventions are indentified and past conventions are rendered absolete 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.

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 ~~the~~ ~~s~~ 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-- Some-times style guide recommendations just aren't applicable. when in dout, use your ~~bu~~ best judgment. Look at other examples and decide what look best. And don't hesitate to ask!

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

Some other good reasons to ignore a particular guideline:

1) when applying the guideline would make the code less readable, even for someone who is used to reading code that follow this PEP.

2) To be consistent with surrounding code that also breaks it (maybe for historic ~~res~~ reasons)-- although this is also an opportunity to clean up someone else's mess (in true XP style).

3) Because the code in question predates the introduction of the guideline and there is no other ~~reson~~ 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.

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 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-function-name(
    var-one, var-two,
    Var-three, var-four)

# wrong:

# Arguments on first line
forbidden when not using
vertical ~~aligr~~ alignment.
foo=
 long-funtion-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, ~~vare~~ 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 muliple lines, it's worth nothing that the combination of a two character keyword (i.e if), plus a single spaces, plus an opening parenthesis creates a natural 4-space indent for the subsequent lines of the multilline 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 suit 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
# Supporting syntax
highlighting.
if (this_is_one_thing and
    that_is_another_thing):
    # since both conditions are
true, we can frobnicate.
    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.)

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:

```
my_list = [
     1, 2, 3,
     4, 5, 6,
     ]
result =
some_function_that_takes_argumen
ts(
   '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_argumen
ts(
   '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 alredy 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 it 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 ~~wramic~~ wrapping at all.

Some teams strongly prefer a longer line length. For code maintained exclusively or primarily by a team that can reach agreemeat on this issue, it is okay to increase the line length limit ~~to~~ up to 99 characters, provided that comments and docstrings are still wrapped at 72 characters.

The python Standard library is conservative and re-quires 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 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 implict continuation before python 3.10, so backslashes where acceptable for that case:

```
with
 open('/path/to/some/file/you/want
 /to/read') as file_1, \

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

(see the previous discussion on multiline if-state-ments for further throughts on the indentation of such multiline with-statements.)

Another such case is with assert statements.

Make sure to indent the continued line appropriately.