FYIT=169 Hafiz Mansuri

Assignment

PEP 8 - : Style Guide for python code

- Introduction

This document gives coding conventions for the python code comprising the code comprising the standard library in the main python distribution the comparision for the coode in the (implementation of python

PEP8 and PEP257 (Dosting) Documen-t conventions) were adapted from Guido original python style guide

This style guide wolves overtime as additional conventions are rendered absolute by changes in the languages itself

* A foolish consistency is the hobgoblin of little minds

One of guides key insight is that code is read much more often than it is written The guidelines provided here are intended to improve the guidelines of code & make

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.

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 clearly distinguish itself as a continuation line.

correct:

Aligned with opening delimiter
foo =
long_function_name(var_one,
var_two,

spirit 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 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 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 follows this PEP.

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

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 alignment
foo =
long_function_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)

```
# 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 procedure
      do_something()

# Add some extra indentation on
  the conditional continuation
  line:
  if (this_is_one_thing
          and
      that_is_another_thing):
      do_something()
```

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_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

FYI 1=1 69

HARIZ MARISMI

The space continuation is optional for continuation lines.

Optional

Hanging indents may be indented to other than 4 spaces (e.g. 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 is worth nothing that across multiple lines it is worth nothing that the combination of a two character keyword i.e. plus a single space, plus an opening parenthesis creates a natural space Main of the subsequent lines at he the multiple conditions

# no extra indication
if (this_is_one_thing &
    this_is_another_thing)
    do_something();

PEP 8 has
limiting the suggested editor window width
makes it possible to have several files open
side by side and work well when using code
review tools that present the two versions
in adjacent panes.

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
by using Python's implied
inside parentheses, brackets
long lines can be broken

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',
)
```

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 (docstrings or comments), the line length should be limited to 72 characters.

FUTURE

wrapping expressions in parantheses. These should be used in preference to using a ba 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())
```

(see the previous discussion on multiline if-statement 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.

```
# correct i
# 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 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 (e.g. a set of dummy implementations).

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

FYI 1 = 10^1

HAFIZ MANSU:

line break before or after a binary operator

for decades the recommended style was to break after binary operators first But this can hurt act scatered across different columns on the screen an operand find onto the previous on the screen and each operators is moved away from it operand and onto the previous line

H wrong

operators at far away from the operands

income = (gross - wages + taxable - interest + dividends - qualified - dividends - trade deduction - student loan - interest

To step solve this readability problem, mathematicians and other publishers follow the opposite convention Donald Knuth

Imports

FYIT 109 hafiz

- Imports should usually be on seperate 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

A blank line is must between each group of imports.

Python allows the control-L (i.e. ^L) form feed character as whitespace, many tools treat these characters as page separators, so may use them to separate pages of related sections of your file. Note, some editor 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 a encoding declaration.

In the standard library, non-UTF-8 encodings should be used only for test purposes. Use non-ASCII characters sparingly, preferably only to denote place 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.)

Open source projects with a global audience are encouraged to adopt a similar policy

F4TT 109 hafiz

- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better messages) if the import system is incorrectly configured (such as when a directory inside a pakag 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.

- 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
```


Scanned by TapScanner

EYIT 109 hafiz

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 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:

```
""" This is the example module.

This module is stuff
"""

from __future__ import
barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'cardinal biggles'

import os
import sys.
```

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

import myclass
import foo.bar.yourclass

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 as part of a public API (for example, overwriting a pure Python implementation of an 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.

FYIT 109-hafiz

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 in PEP257

Whitespace in Expressions & statements

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

correct:

```
spam(ham[1], {eggs:2})
```

Wrong:

```
spam( ham[1], { eggs: 2 } )
```

```
ham[lower:upper],
ham[lower:upper:],
ham[lower::step]
ham[lower+offset:
upper+offset]
ham[:upper_fn(x):
step_fn(x)], ham[::
step_fn(x)]
ham[lower+offset:upper+
offset]

#wrong
ham[lower+offset:upper+
offset]
ham[1:9], ham[1:9], ham[1:9
:3]
ham[lower::upper]
ham[::upper]
```

- Immediately before the open parenthesis that starts the argument list of a function call.

```
# correct:
spam(1)

# wrong:
spam (1)
```

- Between a trailing comma and following close parenthesis:

# correct:
foo = (0,)

# wrong:
bar = (0.)

- Immediately before a comma, semicolon, or colon:

correct:
if x == y: print (x,y); x,y = y,x

wrong:
if x == y : print (x, y) ; x ,
y = y, x

- However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority) 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:

correct:
ham[1:9], ham[1:9:3],
ham[:9:3], ham[1::3],
ham[1:9:]

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

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

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

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

```
# correct:
x = 1
y = 2
long_variable = 3

# wrong:
x             = 1
y             = 2
long_variable = 3
```

- Avoid trailing whitespace anywhere because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker. Some editors don't preserve it and many projects (like CPython itself) have pre-commit hooks that reject it.

Function annotations: PEP 484 hafiz

correct:
def munge(input: AnyStr): ...
def munge() -> PosInt: ...

wrong:
def munge(input: AnyStr) : ...
def munge() -> PosInt : ...

: 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:

# correct:
def complex(real, imag=0.0):
return magic(r=real,
i=imag)

# wrong:
def complex(real, imag = 0.0):
return magic(r=real, i=
imag)

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

correct:
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: ...

FYIT-109 hafiz

- Always surround these binary operator with a single space on either side: assignment (=), augmented assignment (+=, -= etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not) boolean: (and, or, not).

- If operators with different priorities are used consider adding whitespace around the operators with the lowest priority(ies). use your own judgement, however, never use more than one space, and always have the same amount of whitespace on both sides of binary operator:

```
# correct:
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)

# wrong:
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

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

```
AnyStr= None, limit: 1000):

#wrong!
def munge (input: AnyStr=None):
    ...
def munge (input: AnyStr, limit
    =1000): ...
```

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

```
#correct:
if foo=='blah':
    do_bh_blah_thing()
    do_one()
    do_two()
    do_three()
```

Rather not:

```
#wrong!
if foo=='blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- While sometime it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

FAIL 102 half

```
#wrong
if foo == 'blah':
do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
# wrong:
if foo == 'blah':
do_blah_thing()
use: do_non-blah-thing()

try: something()
finally: cleanup()

do_one(); do_two();
do_three(long, argument,

list, like, this)

if foo == 'blah': one();
two(); three()
```

Exception Names

Because exceptions should be classes the same naming convention applies here. However you should use the suffix "Error" on your exception names (if the exception actually is an error).

Designing for inheritance

Always decide whether a class methods and instance variables (collectively: "attributes") should be public or non public. If in doubt, choose non variable public; its easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

We don't use them the term "private" here, since no attribute is really private in python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by base class.