Name:- Vikas Kumar Komavel
Roll No:- 41
Std:- FYIT

Ass

It platforms, Tools and practices

* 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 PEP 257 were adapted from Guido's original Python Style Guide essay, with some additions from Barry's style guide. 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.

* A Foolish Consistency is the Hobgoblin of Little Minds

Ans One of Guido's key insights is that code is read much more often 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 inconsistent - sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. 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 although this is also an opportunity to clean up someone else's mess

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 again wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent [1]. 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 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)

The 4-space rule 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 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 suite inside the if-statement. Acceptable options in this situation include, but are not limited to:

```
# No extra indentation.
if (this_is_one_thing and
    than_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.
```

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'
    )
```

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. ~~Period~~

Python disallows mixing tabs and spaces for indentation.

* 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. Here the eye has do extra work to tell which items are added and which are subtracted:

```
# wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

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)

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

(i) 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-liner

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

Python accepts the control-L form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related pages sections of your file. Note editors and web-based code viewers may not recognize control-L as a feed and will show another glyph in its place.

* Source File Encoding
Code in the core python distribution always use UTF-8, and should not have an encoding declaration.
In the standard library, non-UTF-8 encoding should 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 ~~source~~ (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.

* 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.

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 if the import system is incorrectly configured.

```
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
```

If this spelling causes local name clashes, then 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 interface as part of a public API

When republishing names this way, the guidelines below regarding public and internal interfaces still 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 docstring but before any imports 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 does stuff.
"""

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'

import os
import sys
```

* 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 back-slashes in the string. It improves readability.

for triple-quoted strings, always use double quote 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})

# 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 colon:

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

- ~~Never~~ 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:]
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)
```

- 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
```

★ Other Recommendations.

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing : e.g. 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.

- Always surround 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 the lowest priority(ies). 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:

```
# 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. (See function Annotations below for more about function annotations):

```
# 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: 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_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

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

- While sometimes 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:

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()
else: do_one_other_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

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

* 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:

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

# Wrong:
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/brace on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter

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

# wrong:
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

Block comments generally consist of one or more paragraphs 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, except after the final sentence

Ensure that your comments are clear and easily understandable 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 (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. 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:

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

But sometimes, this is useful:

```
x = x + 1                # compensate for border
```