PEP 8 – Style Guide for Python Code

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

```
# 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)
```

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.

```
# 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()
```

For list

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

OR

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Maximum Line Length

Limit all lines to a maximum of 79 characters.

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())
```

Line Break Before or After a Binary Operator

```
# 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.

Method definitions inside a class are surrounded by a single blank line.

Source File Encoding

Code in the core Python distribution should always use UTF-8,

Imports

Imports should be grouped in the following order:

Standard library imports.
Related third party imports.
Local application/library specific imports.

Absolute imports

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

from M import * does not import objects whose names start with an underscore.

Whitespace in Expressions and Statements

```
# Correct:
spam(ham[1],  {eggs: 2})
# Correct:
foo = (0,)
# Correct:
if x == 4: print(x, y); x, y = y, x
 Correct:
x = 1
y = 2
long_variable = 3
```

Assignment (=)
Augmented assignment (+=, -= etc.)
Comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not),
Booleans (and, or, not).

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)
```

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

```
def munge(input: AnyStr): ...
def munge() -> PosInt: …
```

Don’t use spaces around the = sign when used to indicate a keyword argument

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

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

```
# Correct:
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): …
```

When to Use Trailing Commas

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

Comments

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).

Documentation Strings

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

For single line

```
"""Return an ex-parrot."""
```

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.

Package and Module Names

Modules should have short, all-lowercase names. Underscores

can be used in the module name if it improves readability

Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

Class Names

Class names should normally use the CapWords convention.

Type Variable Names

Names of type variables introduced in PEP 484 should normally use CapWords preferring short names

Function and Variable Names

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

Function and Method Arguments

Always use **self** for the first argument to instance methods.

Always use **cls** for the first argument to class methods.

Method Names and Instance Variables

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

Python mangles these names with the class name: if class Foo has an attribute named __a, it cannot be accessed by Foo.__a. (An insistent user could still gain access by calling Foo._Foo__a.)

Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include MAX_OVERFLOW and TOTAL

Programming Recommendations

Comparisons to singletons like None should always be done with **is** or **is not**, never the equality operators.

Writing **if x** when you really mean **if x is not None**

When implementing ordering operations with rich comparisons, it is best to implement all six operations (__eq__, __ne__, __lt__, __le__, __gt__, __ge__) rather than relying on other

code to only exercise a particular comparison.

__eq__(self,other)	x == y,
__ge__(self,other)	x >= y,
__gt__(self,other)	x > y,
__le__(self,other)	x <= y,
__lt__(self,other)	x < y,
__ne__(self,other)	x != y

Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier:

```
# Correct:
def f(x): return 2*x
# Wrong:
f = lambda x: 2*x
```

Class naming conventions apply here, although you should add the suffix “Error” to your exception classes if the exception is an error. Non-error exceptions that are used for non-local flow control or other forms of signaling need no special suffix.

Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

```
# Correct:
if foo.startswith('bar'):
```

Object type comparisons should always use isinstance() instead of comparing types directly:

```
# Correct:
if isinstance(obj, int):
# Wrong:
if type(obj) is type(1):
```

For sequences, (strings, lists, tuples), use the fact that empty sequences are false:

```
# Correct:
if not seq:
if seq:
```

Don’t compare boolean values to True or False using ==:

```
# Correct:
if greeting:
# Wrong:
if greeting == True:
```

Use of the flow control statements **return/break/continue** within the finally suite of a **try...finally,** where the flow control statement would jump outside the finally suite, is discouraged. This is because such statements will implicitly cancel any active exception that is propagating through the finally suite:

Variable Annotations

Annotations for module level variables, class and instance variables, and local variables should have a single space after the colon

There should be no space before the colon.

If an assignment has a right hand side, then the equality sign should have exactly one space on both sides:

```
# Correct:

code: int

class Point:
    coords: Tuple[int, int]
    label: str = '<unknown>'
```