PEP-8

⊛ INTRODUCTION :-

PEP is an abbreviation form of python enterprise proposal. Writting code with proper logic & a key factor of programming, but many other important factors can affect the code's quality. PEP 8 is a document that provides various guidelines to write readable in python. It describes how developer can write beautiful codes.

The main aim of PEP is to enhance the readability and consistency of code.

⊛ CODE LAY-OUT :-

⇒ INDENTATION:

Uses 4 spaces per indentation level.
Continuation lines should align wrapped element either vertically using python's implicit line joining inside parenthesis, brackets and braces, or using a hanging indent.
When using a hanging indent following should be considered; there should be no arguments on first line and further indentation should be used to clearly distinguish itself as a continuation line.

Example:

```
foo = function_name(var1, var2, var3,
                    var4)
```

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 two character keyword, plus a single space, plus an opening parenthesis creates a natural 4-space indent for subsequent lines of multiline conditional.
The closing brace/bracket/parenthesis on multiline constructs may ~~ethier~~ either line up under the first non-whitespace character of last line of list or it may be lined up under the first character of line that starts multiline construct.

⇒ TABS OR SPACES :

Spaces are 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 :
Limits all lines to a maximum of 79 characters. Limiting the required editor window width makes it possible to have several files open by side, and works well when using code review tools that present the two version in adjacent columns.
The limits are chosen to avoid wrapping in editors with the window width set to 80, even if tool places a marker in final column when wrapping lines.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parenthesis, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parenthesis. Backlashes may still be approprate at times.

⇒ LINE BREAK BEFORE OR AFTER BINARY OPERATOR :
The recommended style was to break after binary operators. But this can hurt readibility in two ways : the operators tend to get scattered across different columns on screen, and each operator is moved away from its operand and onto previous line.
In python code, it is permissible to break before or after a binary operator, as long as convention is consistent locally.

⇒ 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.

⇒ SOURCE FILE ENCODING :
Code in core python distribution should always use UTF-8, and should not have an encoding declaration. In standard library, non-UTF-8

encodings should be used only for test purposes
Use non-ASCII characters sparingly, preferably
only to denote places and human names. All
identifiers in python standard library must use
ASCII-only ~~fnd~~ identifiers, and should use
english words wherever feasible. Open source
projects with a global audience are encouraged
to adopt a similar policy.

→ IMPORTS:
Imports should usually be on separate lines.
They are always put at top of file, just after
any module comment and docstrings, and
before module globals and constants.
Imports should be grouped in following order:
1) Standard library imports
2) Related third party imports
3) Local application / library specific imports.
Absolute imports are recommended, as they
are usually more reable and tend to be better
behaved if the import system is incorrectly
configured.
Standard library code should avoid complex
package layouts and always use absolute
imports.

⇒ MODULE LEVEL DUNDER NAMES:
Module level "dunder" (names with two
leading and two trailing underscores) such
as __all__, __author__, __version__, etc
should be placed after module docstring but

before any import statements except from __future__ imports. Python mandates that future-imports must appear in module before any other code except docstrings.

✱ STRING QUOTES :-

In python, single-quoted strings and double-quoted strings are same. This PEP does not make a recommendation for this.
When a string contains single or double quote characters, use the other one to avoid backslashes in the string. It improves readability.

✱ WHITESPACES IN EXPRESSIONS & STATEMENTS :-

⇒ PET PEEVES :

Avoid extraneous whitespaces in -

- immediately inside parenthesis, brackets or braces
- between trailing comma & following close parenthesis
- immediately before a comma, semicolon or colon
- immediately before open parenthesis that starts argument list a function call
- immediately before open parenthesis that starts an indexing or slicing
- more than one space around an assignment operator to align it with another.

⇒ OTHER RECOMMENDATIONS :

- Avoid trailing whitespace anywhere. Because it's

usually invisible, it can be confusing. Some editors don't preserve it and many projects have pre-commit hooks that reject it.

- Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, -=), comparisons (==, <, >, !=, <>, <=), Booleans (and, or, not)
- Function annotations should use the normal rules for colons and always have spaces around the → arrow if present.
- When combining an argument annotation with a default value, do use spaces around the = sign.
- Compound statements (multiple statements on same line) are generally discouraged.

⊛ WHEN TO USE TRAILING COMMAS :-

Trailing commas are usually optional, except they are mandatory when making a tuple of one element.
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 a line by itself, always adding a trailing comma, and add close parenthesis / bracket / brace on next line.

⊛ COMMENTS :-

Comments that contradict 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 1st word should be capitalised, unless it is an identifier that begins with a lowercase letter.
Block comments generally consist of one or more paragraphs built out of complete sentences with each sentence ending in a period.
Ensure that your comments are clear and easily understandable to other speakers of language you are writing in.

⇒ BLOCK COMMENTS :

Block comments generally apply to some code that follows them, and are indented to same level as that code. Each line of a block comment starts with # and a single space.
Paragraphs inside a block comment are separated by a line containing a single #.

⇒ INLINE COMMENTS :

An inline comment is a comment on the same line as a statement. Inline comments should be separated by atleast two spaces from statement. They should start with # and single space.
Inline comments are unnecessary and in fact distracting if they state the obvious.

⇒ DOCUMENT STRINGS :
Write docstrings for all public modules, functio
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 appea
after the def line.
It describes good docstring conventions. The
""" that ends a multiline docstring should
be on a line by itself.

⊛ NAMING CONVENTIONS :-

The naming conventions of python's library
are bit of mess, so we'll never get this
completely consistent - nevertheless, are
currently recomended naming standards.
New modules and packages should be
written to these standards, but where an
existing library has different style, internal
consistency is preferred.

⇒ OVERIDING PRINCIPLES :
Names that are visible to the user as public
parts of API should follow conventions that
reflect usage rather than implementation.

→ DESCRIPTIVE : NAMING STYLES :
There are a lot of different naming styles. It
helps to be able to recognise what naming
style is being used, independently from what

they are used for.
Following naming style are commonly distinguish-
- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords
- mixedCase
- Capitalized_Words_With_Underscores.

⇒ PRESCRIPTIVE : NAMING CONVENTIONS :

> Names to avoid –
Never use the character 'l', 'O', or 'I' as single character variable names.
In some fonts, these characters are indistinguishable from 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 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.

When an extension module written in C or C++ has an accompaying python module that provides a higher level interface, the C/C++ module has a leading underscore.

> Class names -
Class names should normally use CapWords convention.
The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.
Note that there is a separate convention for builtin names: most builtin name are single words, with the CapWords convention used only for exception names and builtin constants.

> Type Varrable names -
Names of type varrables introduced in PEP 484 should normally use CapWords preferring short names: T, AnyStr, Num. It is recommended to add suffixes _co or _contra to the variable used to declare covariant or contravariant behavior correspondingly.

> Exception names -
Because exception should be classes, the class naming convention applies here.

However, you should use the suffix "Error" on your exception names. (if the exception actually is an error).

> Global variable names -
The conventions are about the same as those for functions.
Modules that are designed for use via from M import * should use the __all__ mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

> 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.
mixedCase is allowed only in contexts where that's already the prevailing style, to retain backwards compatibility.

> Function and method arguments -
Always use self for first arguments to instance methods.
Always use cls for first arguments to class methods.
If a function argument's name clashes with

a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class_ is better than cls.

> Method names & 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.
Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

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

> Designing for inheritance -
Always decide whether a class's methods and instances variables (collectively:

"attributes") should be public or non-public. If in doubt, choose non-public; its easiers 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.
Another category of attributes are those that are part of the "subclass API". 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 your base class.
Here are pythonic guidelines:
- Public attributes should have no leading underscores.
- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling.
- For simple public data attributes, it is best to expose just the attribute name, without

complicated accessor/mutator methods. Keep in mind that python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior.

- If your class is intended to be subclassed, and you have attributes that you do not want subclass to use, consider naming them with double leading underscores and no trailing underscores. This invokes python's name mangling algorithm, where the name of class is mangled into attribute name. This helps avoid attribute name collisions should subclass inadvertently contain attributes with the same name.

⇒ PUBLIC AND INTERNAL INTERFACES:

Any backwards compatibility guarantees apply only to public interfaces. Accordingly, it is important that users be able to clearly distinguish between public and internal interfaces.

Documented interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces exempt from usual backwards compatibility guarantees. All undocumented interfaces should be assumed to be internal.

To better support introspection, modules should explicitly declare the names in their public API using the __all__ attribute. Setting

-all- to an empty list indicates that the module has no public API.
Even with -all- set appropriately, internal interfaces (packages, modules, classes, functions, attributes or other names) should still be prefixed with a single leading underscore.
An interface is also considered internal if any containing namespace (package, module or class) is considered internal.
Imported names should always be considered an implementation detail.

✱ PROGRAMMING RECOMMENDATIONS :-

- Code should be written in a way that does not disadvantage other implementations of python. For eg, do not rely on CPython's efficient implementation of inplace string concatenation for statements in form a += b or a = a + b. This optimization is fragile even in CPython and isn't present at all in implementations that don't use refcounting. In performance sensitive parts of library, the ''.join() form should be used instead. This will ensure that concatenation occurs in linear time across various implementations.

- Comparisons to singletons like None should always be done with is or is not, never the equality operators.
Also, beware of writing if x when you really

mean if x is not None — eg. when testing whether a variable or argument that defaul to None was set to some other value. The other value might have a type that could be false in a boolean context.

- Use is not operator rather than not ... is. While both expressions are functionally identical, the former is more readable and preferred.

  correct —
  if foo is not None:

  Incorrect —
  if not foo is None:

- 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

  incorrect —
  f = lambda x: 2*x

- When catching exceptions, mention specific exceptions whenever possible instead of using a bare except:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

- When catching operating system errors, prefer the explicit exception hierarchy introduced in python 3.3 over introspection of err no values.
- Additionally, for all try/except clauses, limit the try clause to absolute minimum amount of code necessary.

```
correct -
    try:
        value = collection[key]
    except KeyError:
        return key_not_found(key)
    else:
        return handle_value(value)
incorrect -
    try:
        return handle_value(collection[key])
    except KeyError:
        handle_value()
        return key_not_found(key)
```

- When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.

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

```
correct -
    if foo.startswith('bar'):
```

incorrect —
if foo[:3] == 'bar':

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

correct —
if isinstance(obj, int):
incorrect —
if type(obj) is type(1):

- For sequences, (strings, lists, tuples), use fact that empty sequences are false.

correct —
if not seq:
if seq:
incorrect —
if len(seq):
if not len(seq):

- Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors will trim them.

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

correct —
if greeting:
incorrect
if greeting == True:

→ FUNCTION ANNOTATIONS:

- Function annotations should use PEP 484 syntax
- The experimentation with annotation styles that was recommended previously in this PEP is no longer encouraged.
- The python standard library should be conservative in adopting such annotations, but their use is allowed for new code and for big refactorings.
- For code that wants to make a different use of function annotations it is recommended to put a comment of form:#
- Like linters, type checkers are optional, separate tools. Python interpreter by default should not issue any messages due to type checking and should not alter their behavior based on annotations.

→ VARIABLE ANNOTATIONS:

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

- Although the PEP 526 is accepted for python 3.6 the variable annotation syntax is the preferred

syntax for stub files on all versions of
python (see PEP 484 for details).