IT Tools

PEP Stands for Python Enhancement proposal. This document gives coding conventions for the Python code comprising the standard library in the main python distribution.

Introduction :-

This document gives coding and PEP 257 (Docstring Conventions) 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. 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 PEP20 Says, "Readability Counts". A Style guide is about consistency. Consistency with this Style is important. Consistency within a projects is more important. Consistency within one module or function is the most important.

Page No.:
Date: / /

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 parantheses, 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. The 4-space rule is optional for continuation line.

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 nothing that the combination of a two character keyword (ie. 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 (or whether) 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
    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 fabricate.
  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-white space 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',
    )
```

Tabs or Spaces?

Spaces are the prefferred 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 indentations

Date / /

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-liners (e.g. a set of dummy implementations)

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 sections of your file. Note, some editors and web-based code viewers may not recognize Control-L as a form feed and will show another glyph in its place.

Code in the core Python distribution should always use UTF-8, and should not have an encoding declaration.

Page No.:
Date: / /

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 (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a 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:

Date / /

from myclass import Myclass
from foo.bar.yourclass import
yourclass

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

import myclass
import foo.bar.yourclass

and use "myclass.Myclass" and "foo.bar.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 implentation 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 interface still apply.

Module Level Dunder Names

Module level "dunder" (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 does stuff.
"""

from __future__ import
    barry_as_FLUE

__all__ = ['a', 'b', 'c']

__version__ = '0.1'

__author__ = 'Cardinal Biggles'

import os
import sys
```

Date / /

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 (never alter the case of identifiers!).

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

It 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 comments starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a blocks comment are separated by a line containing a single #.

Inline Comments

use inline comments sparingly.

An inline comment 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

x = x + 1
compensate for border
```

Page No.
Date / /

Documentation Strings

Conventions for writing good documentation strings (a.k.a "docstrings") are immortalized in PEP 257.

* Write docstrings for all public modules, functions, 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 appear after the def line.

* PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multline docstring should be on a line by itself:

```
"""Return a foobang

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

* For one liner docstrings, please keep the closing """ on the same line:

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

class Names

class Names should normally use the capwords convention

The naming convention for functions may be used instead in case where the interface is documented and used primarily as a callable

Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the capwords convention used only for exception names and builtin constants

Type Variable Names

Names of types variables 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 variables used to declare covarient or contravarient behavior correspondingly:

```
from typing import TypeVar

VT_co = TypeVar('VT_co',
covariant = True)
KT_contra = TypeVar('KT_contra',
contravariant = True)
```

Exception Names

Because exceptions 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

(let's hope that these variables are meant for use inside one module only.) 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 underscore as necessary to improve readability.

Variables names follow the same convention as function names.

Mixed case is allowed only in contexts where that's already the prevailing style to retain backwards compatibility.

Functional and Method Arguments

Always use self for the first argument to instance methods

Always use cls for the first argument 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. (Perhaps better is to avoid such clashes by using a synonym.)

Method Names and Instances Variables

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

Use one of the 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.

Date / /

Programming Recommendations

Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, iron Python, cython, Psyco and such).

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

```
# Correct:
if foo is not None:


# wrong:
if not foo is None:
```