PRACTICAL 7

Implementing coding practices in python using PEP8

➢ **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 (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.

➢ **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 used to clearly distinguish itself as a continuation line:

```
# 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)
```

- **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.

- **Source File Encoding**

Code in the core Python distribution should always use UTF-8, and should not have an 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 places and human names. If using non-ASCII characters as data, avoid noisy Unicode characters like z̯̯͡a̧͎̺l̡͓̫g̹̲o̡̼̘ 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.

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:
- from myclass import MyClass
- from foo.bar.yourclass import YourClass

  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.

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 does stuff.
"""

from __future__ import barry_as_FLUFL

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

import os
import sys
```