St took

Fyit - 139

Assignment - 2

Joy. Mothi

- What is python PEP 8 ?

→ PEP 8, sometimes spelled PEP 8 or PEP-8, is a document that provides guidelines and practices on how to write python code .... A PEP is a document that describes new feature proposed for python and documents aspects of python, like design and style, for the community

- Introduction

This document gives coding convections for the python code comprising the standard library in the main python distribution. Please see the companion informational PEP describing style guideline for the C code in the C implementation of python. This style guide evolves over time as additional convention are identified and past convention 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 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:

```
# 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)
```

The closing brace/ bracket/ parenthesis on multiple 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',
)
```

- Tabs and 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 78 characters.
For flowing long blocks of text, with fewer structural restrictions (docstring or comments), the line

should be limited to 72 characters.
Limiting the required editor window width makes it possible to have several files open side by side, and works well when using code review tools that present the two versions in adjacent columns.

- Blank lines
Surround top-~~lines~~ level functions 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 seperate groups of related functions. Blank lines may be omitted betwen a bunch of related one-liners (e.g. a set of dummy implementations.

- 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 encodings should be used only for test purposes. Use of non-

ASCII characters as data, avoid noisy Unicode characters like zalgo and order marks.

All identifiers in 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.)

Imports

* Imports should usually be on seperat lines:

# Correct
import os
import sys

* Imports are always put at the top of t file, just after any module comment and docstrings, and before module globals and constrants.

Imports should be grouped in th following groups

i. Standard library imports.
ii. Related third party imports
iii. local application/library speci imports.
iv.

- Module level dunder Names
  Module level "dunders" such as __all__, __author__, __version__, etc. should be placed after the module docstring but before any import statements except 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_FLU
__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 backslashes in the string. It improves readability.

- 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 later in parentheses:

Correct:

```
FILES = ('setup.cfg',)
```

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date with code changes!

Comments should be complete sentences. The first word should be capitalized unless it is an identifier that

begins with a lower case latter (never alter the case of identifiers!)
Block comments generally consist of one or more paragraphs build out of complete sentences, with each sentences ending in a period.
You should use two spaces after a sentence-ending period in multi-sentence comments, except after the final sentence.
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 comment starts with a # and a single space

- Naming Conventions
The naming conventions of Python's library are a bit of mess, so we'll never get the completely consistent - nevertheless, here are currently recomended naming

standards. New modules and packages should be written to these standards, but where an existing library has a different style, internal consistency is preffered

- Overriding principle
  Names that are visible to the users as public parts of the 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 :-

  The following naming styles are commonly distinguised:

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores

- UPPERCASE
- UPPER_CASE_WITH_UNDERCASE
- Capitalized Words
- mixed Case
- Capitalized_words_with_underscores.

- Class Names
  Class names should normally use the CapWords convention.
  The naming convention for funct may be used in cases where the interface is documented and prima as a callable.

- function and Variable name
  function names should be lowercase, with words seperated by underscores as necessary to improve readability.

- functions and Method Arguments.
  Always use self for the first argument to instance methods.
  Always use cls for the first argument to class methods.

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

- Designing for Inheritance.

  Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

- 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 the usual backwards compatibility guarantees. All undocumented ~~interfaces~~ interfaces should be assumed to be internal.

- Programming Recommendations:-
  Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).

- 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
  ```

A good rule of thumb is to limit use of bare 'except' clauses to two cases:-

i) if the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occured.

ii) if the code needs to do some clean up work, but then lets the exception propogate upwards with raise .... Finally can be a better way to handle this case

- when catching operating system errors, prefer the explicit exception hierarchy introduced in Python 3.3 over introspection of error values.

- Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs.

Correct:

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

return handle_value(value)

- Don't write string literals that rely on significant whitespace. Such trailing whitespace is visually indistinguishable and some editors (or, more recently, reindent.py) will trim them.
- Don't compare boolean values to True or False using ==:

Correct:

if greeting:

Wrong:

if greeting == True:

Worse

Wrong:

if greeting is True:

- function Annotations

  With the acceptance of PEP 484, the style rules for function annotations have changed.

- function annotations should use PEP 484 syntax

- The experimentation with annotation styles that was recommended previously in this PEP is no longer encouraged.

- However, outside the stdlib, experiments within the rules of PEP 484 are now encouraged. for example, marking up a large third-party library or application with PEP 484 style type annotations, reviewing how easily it was to add those annotations, and observing their presence increases code understandability.

The Python standard library should be conservative in adopting such annotations, but their use is allowed for new code and for big refacto

- for code that wants to makes a different use of function annotation it is recommended to put a comment of the form:

type : ignore

near the top of the files; this tells type checkers to ignore all annotations.

- like linters, type checkers are optional, separate tools. python interpreters by default should not issue any messages due to type checking and should not alter their behaviour based on annotation.

- users who don't want to use type checkers are free to ignore them. However, it is expected that users of third party library packages may want to run type checker over those packages. For this purpose PEP 484 recommends the use of stub files :- pyi files that are read by the type checker in preference of the corresponding .py files.

Variable Annotations

PEP 526 introduced variable annotations. The style recommendations for them are similar to those on function annotations described above:-

- Annotations for module level variables, class and 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>'
```

Wrong:

code: int # No space after colon.

code : int # space before colon.

class Test:

result: int.=0 # No spaces around equality sign.

- 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