FyIT/Mohammad Sadik/38

IT Tools and Practices

* PEP in python *

1) What is PEP?

→ PEP stands for Python Enhancement Proposal. A PEP is a design document providing information on to the Python community, or describing a new feature for Python or its processes or environment. The PEP should provide a concise technical specification of the feature and a rationale for the feature.

* Code Lay-out

* use 4 spaces per indentational 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.

```
# 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, tw
var_three.
        var_four):
    print (var_one)
# Hanging indents should add a level.
foo = long_function_name (var_one, var_t
                          var_three, var_four)

# wrong:

# Arguments on first line for hidden when no
using vertical alignment.

Foo =
long_function_name (var_one, var_two, var_th
                    var_four)

# further indentation required as indentatio
is not distinguishable.
def long_function_name (var_one, var_two, va
                        three, var_four):
    print (var_one)
```

Page No. 3
Date

* The 4-space rule is optional for continuation lines.

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 character keyword (ie. if) plus a single, 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-state most, which would also naturally be indented to -4 spaces.

This PEP takes no explicit position on how 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 indetation.

```
if (this - is - one thing and that is - another
thing do - something ()
```

Page No. 4
Date

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 condition continuation line.

```
if (this - is - one - thing
        and
    that - is - another - thing):
        do - something ()
```

The closing brace/ braced/ paranthesis 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,
    ]
```

Page No. 5
Date

```
result =
some_function_that_takes_arguments('a', 'b', 'c',
                                   'd', 'e', 'f')
```

or it may be lined up under the first character of the line that starts the multiline construct, 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 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.

Page No. 6
Date

Limit all lines to a maximum of 79 charecters.

For flowing long blacks of text with fewer structural restrictions, the line length should be limited to 72 charecters.

Limiting the required editor window width makes it possible to how several files open side by side, and works well when using code review tools that present the two various in adjacent columns.

The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chose to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final colum when us apping lines. Some web based tools may not after dynamic line wrapping at all.

* Source file Encoding

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

All identifies in the Python standard library must use ASCII-only identifies, and should use English words feasible

Page No. 7
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) Stadard library imports
2) Related third party imports.
3) Local application / library specific imports.

* Absolute imports are recommended, as they are usually more readable and tend to be better behaved if the imports system is

Page No. 8
Date

incorrectly configured.

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

However, explicit relations imports are an acce
alternative to absolute imports, especially whe
dealing with complex package layouts where
using absolute imports would be unnecessa
verbose.

from. import sibling
from. sibling import
example

Standard library code should avoid complex
package layout and always use absolute
imports.

when importing a class from a class-containing
module, it's usually okay to spell this;

from my class import
my class
from foo.bar.your class
import your class

Page No. 39
Date

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".

* 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 from __future__ import. Python mandates that future imports must appear in the module before any other code except docstrings:

```
"""This is the example
   module.
This module class stuff.
"""
from __future__ import
barry_as_FLUFL
__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'cordinal
Biggles'
import os
import sys
```

Page No. 11
Date

* 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.

For triple-quoted string, always use double quote characters to be consistant with the docstring convention in PEP 257.

* Whitespace in Expressions and Statements

Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parantheses, brackets or braces:

```
# Correct:
spam(ham[1], {eggs: 2})
```

Page No. 12
Date

wrong;
Spam (ham[1], {eggs:2})

* Between a trailing comma and a following close parenthesis

# correct:
foo = (1,0,)

# wrong;
bar = (0,)

* Immediately before a comma, semicolon, or colon:

correct;
if x==y: print(x,y); x,y=y,x

wrong:
if x==y; print(x,y); x,y=y,x

* However, in a slice the colon acts like a binary operator, and should have equal amount on either side. In an extended slice, both colons must have the same amount.

of spacing applied.
EXception: when a slice parameter is om, the space is omitted:

correct:
ham [1:9) , ham [1:9:3),
ham [:9:3] , ham [1:,:3),
ham [1:9:]
ham [lower:upper],
ham [lower:upper:],
ham [lower::step]
ham [lower+offset:
upper+offset]
ham[:upper-fn(x):
step-fn(x)], ham[::
step-fn(x)]
ham [lower+offset:upper+offset]

wrong:
ham[lower+offset:upper+offset]
ham [1:9]] , ham[1:9],
ham [1:9:3)
ham [lower::upper)
ham [:upper)

* Immediately before the open parenthesis that starts the argument list of a function all!

Page No. 14
Date

Correct:
spam (1)

wrong:
spam (1)

* Immediately before the open parenthesis that starts an indexing or slicing

correct:
dct['key'] = lst [index]

wrong:
dct ['key'] = lst [Index]

* More than one space around an assignment (or other) operator to align it with another.

# correct:
x = 1
y = 2
long_variable = 3

wrong:

x = 1
y = 2

long_variable = 3

Page No. 15
Date

* Descriptive: naming styles

There are a lot of different naming styles. It helps to be also to recognizing what naming style is being used, independently from what they are use for.

The following naming styles are commonly distinguished:

1) b (single lowercase letter)

2) B (single upper case letter)

3) lowercase

4) lower-case-with-underscores

5) UPPER CASE

6) UPPER-CASE-WITH-UNDERSCORES

7) CapitalizedWords for (CapWords, or CamelCase so named because of the bumpy look of its letters.
This is also sometimes known as StudlyCaps.

Page No. 16
Date

* mixed case (differs from capitalized words by initial lowercase character)

* capitalized_words_with_underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. for example the os.stat() function returns a tuple whose items traditionally have names like st_mode, st_size, st_mtime and soon.

* Single_leading_underscore: "weak" internal use" indicator. Eg. from M import * does not import objects whose names start with an underscore.

* Single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, eg.
  Tkinter.Toplevel(master, class_='ClassName')

* __double_leading_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo; see below)

Page No. 16
Date

* mixed case (differs from capitalized words by initial lowercase character)

* capitalized_words_with_underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. for example the os.stat() function returns a type whose items traditionally have names like st_mode, st_size, st_mtime and soon.

* Single_leading_underscore: "weak" internal use" indicator. Eg. from M import * does not import objects whose names start with an underscore.

* Single_trailing_underscore_: used by convention to avoid conflicts with Python keyword, eg.
  tkinter.Toplevel(master, class_='ClassName')

* __double_leading_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo; see below)

Page No. 17
Date

* __ double - leading - and - trailing - underscore "magic" objects or attributes that live in user - controlled name space. E.g.
__ init __, __ import __ or __ file __ Never invent such names; only use them as documented.

* 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.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

* Exception names

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names.

Page No. 18
Date

* function and variable names

function names should be lowerscore, with words separated by underscores as necessary to improve readability.

variable names follow the same convention as function names

mixedcase is allowed only in contacts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

* function and method arguments.

Always use self for the first argument to class method

If a function argument's name clashes with a reserved key word, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class_ is better than clss.

* method names and instance variables

Page No. 19
Date

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

use one leading underscore only for non-public method and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python's 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.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of __names.

* constants

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

Page No. 20
Date

* Designing for Inheritance:

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

Public attributes are those that you expect related clients of your to use, with your commitment to avoid back ward incompatible changes. Non Public attributes are those are not intended to be used by third parties; you make guarantees that non-Public attributes won't change or even by removed.

With this in mind, here are the Pythonic guidelines:

1) Public attributes should have no leading under-scores

2) If your Public attribute name calling with a reserved keyword, append a Single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling

Page No. 21
Date

Note1:See the argument name reccommandation for class methods.

* for simple public data attributes, it is beet to supose just the attribute na without complicated accesses/mutator me keep in mind that Python provides a easy path to futare enchancement, Sh you find that a simple data attribut needs to grow functional behavior. In that case, use Properties to hid functional implementation behind simple data attribute acces syntax.

Note 1: Try to keep the functional beh side-effect free, although side-effects s ans cocbing are generally fine.

Note2: Avoid using Properties for computation-ally expansive operation: attribute notation makes the caller be that access it (relatively) cheap.

Page No. 22
Date

8)

a) Like linters, type checkers are optional, seperate tools. Python interpreters by default should not issue any message due to type checking and should not affer their behavior based on annotations.

* Variable Annotations

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

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

2) There should be no space before the colon.

3) 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 aro
equality sign.
```

* Although the PEP 526 is accept for Python 3.6, the variable an tion syntax is the preferred syntax for sub files on all versions of Python