

①

IT TOOLS & PRACTICES.

PEP in Python.

(1) What is PEP?

→ PEP stands for Python Enhancement Proposal. A PEP is a design document providing information 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 indentation level.

Continuation lines should align wrapped elements either vertically using Python implicit - line joining inside parenthesis, 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 an continuation line.

(2)

Correct :-

Aligned with opening ~~def~~ delimiter
 foo =
 long - function - name (var-one, var-two,
 var-three, var-four).

Add 4 spaces.

(an extra level of ~~for~~ Indentation to
distinguish arguments from the rest.)def long - function - name (var-one, var-
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)

Wrong :

Arguments on first line forbidden
when not using vertical alignment.
foo =long - function - name (var-one, var-two,
var-three, var-four).# further indentation required as indentation
is not distinguishable.

(3)

```
def long_function_name(var_one, var_two,
                      var_three,
                      var_four):
    print(var_one).
```

* 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, its worth character keyword (i.e. 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-statement, which would also naturally be indented to 4-space. This PEP takes no explicit position on how to further visually distinguish such conditional lines from the nested

(4)

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 frobnicate.
 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 multi-line constructs may either
 line up under the first non-white
 space, character of last line of list, as in

⑤

```
my-list - [
    1, 2, 3,
    4, 5, 6,
].
```

result = some-function-that-takes-arguments ('a', 'b', 'c',
 'd', 'e', 'f',
)

or it may be lined up ~~as~~ under
 the first character of an 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 only soley
 to remain consistence with code that
 is already indented with tabs.

(6)

Python disallows mixing tabs and spaces for indentation.

* Maximum line length.

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.

The default wrapping is most tools disrupts the visual structures of the code, making it more difficult to understand. The limits are chosen to avoid wrapping in editors with the window set to 80, even if the tools places a marker glyph in the final column when wrapping lines. Some web based tools may not offer # dynamics line wrapping at all.

* Source file encoding.

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

④

All identifiers in the python standard library must use ASCII-only identifiers, and should use English words feasible.

* 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 :-

(8)

- (1) Standard 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 import system is incorrectly configured.

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 a complex package layouts where using absolute imports would be unnecessarily verbose e.g-

from . import sibling
 from sibling import
 example

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

⑨

when importing a class from a class-containing module, it's ~~not~~ 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".

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

(10)

""" This is the example module.

This module does stuff

from __future__ import
barry_as_FLUFL

--- all -- = ['a', 'b', 'c']

--- version -- = '0.1'

--- author -- = 'cardinal'

Bingles'

import as
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 backslash in the string. It improves readability.

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

⑪

- * whitespace in expressions and statements.

Pet Peeves

Avoid extraneous whitespace in the following situations :-

- Immediately inside parenthesis, brackets or braces :-

Correct :-

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

Wrong :-

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

- * Between a trailing comma and a following close parenthesis :-

Correct :-

(foo = (0,)

Wrong :-

bar = (0 ,)

- * Immediately before a comma, semicolon or colon :-

(12)

Correct :

```
if x == 4 : print(x,y) : x,
y = y , x
```

Wrong :

```
if x == 4 ; print(x,y) ;
x, y = y , x.
```

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

exception & when a slice parameter is omitted the space is omitted:

correct :

```
ham[1:9], ham[1:9:3],
```

```
ham[89:3], 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].
```

(3)

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 call :-

Correct :-
 spam (1)

Wrong :-
 spam (1)

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

Correct :-
 det ['key'] = 1st. [index]

Wrong :-
 det ['key'] = 1st [index]

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

(14)

Correct :

$$x = 1$$

$$y = 2$$

$$\text{long_variable} = 3$$

Wrong :

$$x = 1$$

$$Y = 2$$

$$\text{long_variable} = 3$$

† Descriptive naming styles.

There are lot of different naming styles.
 It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished :-

- (1) b (single lowercase letter)
- (2) B (single uppercase letter)
- 3) lowercase .
- 4) lower-case-with-underscores
- 5) UPPERCASE .

(15)

6) UPPER-CASE-WITH-UNDERSCORES.

- 7) Capitalized words (or apwords, or Camel case - so named because of the hump & look of its letters. This is also sometimes known as study caps.)
- * 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 so on.

* Single-leading-underscores: weak "internal use" indicator e.g. from `M import *` does not import objects whose names starts with an underscore.

(16)

- * Single - trailing - underscore - Used by convention to avoid conflicts with Python keywords ; eg. :

`tkinter.Toplevel(master,
class = 'class name').`

- * -double - leading - underscore ? When naming a class attribute , invokes name mangling .
(Inside class `FOOBat` ; `_boobie` - comes - `foobar` - `boo` ; see below) .

- * -double - leading - and - trailing - underscore .

"magic" objects or attributes that are user-controlled namespace . e.g.

`__init__` - , `__import__` - or - `file` - . Never invent such names ; only use them as documented .

- * Names to avoid .

Never use the characters '`I`' (lowercase letter) , '`O`' (uppercase letter oh) , or '`I`' (uppercase letter eyes) as single character variable names .

(7)

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 ~~name~~ names.

* 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.

Mixed case is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to certain backwards compatibility.

* Function and Method arguments

Always use self for the first argument to class methods.

(18)

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. This class_ is better than class. (Perhaps better is to avoid such clashes by using a synonym).

* Method Names and Instance Variables

Use the function naming rules: lower-case 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 from with subclasses, use two leading underscores to invoke Python's name mangling rules.

Python mangles the 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 a capital letters with underscores separation words. Examples include MAX_FLOW and Total .

* Designing for Inheritance .

Always decide whether a class's methods and instance variables (collectively ; "attributes") should be public or 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 be removed .

With this in mind , here are the Pythonic guidelines :

(20)

- 1] Public attributes should have no leading underscores.
- 2] 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.

Note 1 : See the argument name recommendation above for class methods.

- * 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 behaviour. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1 : Try to keep the functional behaviour side - effect free, although side - effects such as caching are generally fine.

(21)

Note 2 :- Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access it (relatively) cheap.

* 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 faces exempt from the 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.

(22)

* Even will - all - sit appropriately, internal interfaces (.packages, module, 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.

* Function Annotations :-

- (1) Function annotation should use PEP 484 syntax.
- (2) The experimentation with annotation styles that was recommended previously in the PEP is no longer encouraged.
- (3) However, outside the stdlib, experiments within the rules of PEP 484 are now encouraged. For example, making up a large third party library or application with PEP 484 style type annotations.

(23)

- 4) The Python standard library should be conservative in adopting such annotations but their use is allowed for new code and for big refactorings.
- 5) For code that wants to make a different use of function annotations it is recommended put a comment of the form :
- ```
type ignore .
```
- 6) 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 annotations.

### \* Variable Annotations :-

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

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

(24)

- 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 around  
equality sign.

(5)

- \* 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.