Page No.
Date

IT Tools & Practices

* PEP in Python * (1)

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 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. (2)

```
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)
```

Wrong:

Arguments on first line forbidden when not using vertical alignment.

```
foo =
long_function_name (var_one, var_two,
                    var_three , var_four)
```

(3)

Further indentation required as indentation is not distinguishable.

```
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, it's 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-spaces.

(4)

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 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()
```

(5)

The closing brace/bracet/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',
                                    )
```

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',
                                    )
```

(6)

* 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, the line length 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.

The default wrapping in most tools disrupts the visual structure of the code, making it more difficult to understand. The limits are chosen

(7)

to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic 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.

All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible.

* Imports.

Imports should usually be on separate lines:

```
# Correct:
import os
import sys

# Wrong:
import sys, os
```

(8)

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.

* 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

(9)

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".

(10)

* 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:

```
""" 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
```

(11)

* 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 strings, always use double quote characters to be consistent with the docstring convention in PEP 257.

* Whitespace in Expressions and statements.

Pet Peeves

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

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

(12)

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:

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 amounts on either side. In an extended slice, both colons must have the same amount

of spacing applied.
Exception: When a slice parameter is omit 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 1 : 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:
dct ['key'] = 1st [index]

Wrong:
dct ['key'] = 1st [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

Page No.
Date

(15)

long_variable = 3

* Descriptive: Naming Styles

There are a 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

6) UPPER_CASE_WITH_UNDERSCORES

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

Page No.
Date

(16)

* mixedCase (differs from CapitalizedWords 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_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

(17)

Consider class FooBar, __boo becomes _FooBar__boo; see below).

* __double_leading_and_trailing_underscore "magic" objects or attributes that live in user-controlled namespaces. 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.
Date

(18)

* 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.

mixedCase is allowed only in contexts 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 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 clss. (Perhaps better is to avoid such clashes by using a synonym).

* Method Names and Instance Variables.

(19)

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

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores too invoke Pythyon'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. (An insistent user could still gain access by calling Foo._Foo—a.) Generally, double leading underscores should be used only too 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 words. Examples include MAX_OVER_FLOW 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 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 backward 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 be removed.

With this in mind, here are the Pythonic guidelines:

1) Public attributes should have no leading underscores.

2) If your public attribute name collides with a reserved keyword, append a single trailing

Page No.
Date

(21)

understo 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 behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

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

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

Page No.
Date

22

* Public and Internal interfaces.

Any backwards compatibility guarantees apply only to public interfaces. Accordingly, it is important that users to 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.

Even with __all__ set appropriately, internal interfaces (packages, modules, 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

Page No.
Date

considered internal.

* Function Annotations.

1) Function annotations should use PEP484 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 PEP484 style type annotations.

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
```

Page No.
Date

24

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 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 variables, 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,
```

Page No.
Date

25

```
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.