SHAIKH ARBAZ ALI

Date
Page

II Tools & Practices

⇒ PEP-8 Coding Practices in Python

PEP stands for Python Enhancement Proposal. This document gives coding conventions for the Python code comprising the standard library in the main Python distribution.

Code lay-out

Indentation

Use 4 spaces per indentation level. Continuation line should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging element. 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.

```
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
# More indentation included to distinguish this 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)
```

When the conditional part of an if-statement is long enough to require that it be written across multiple lines, it's worth nothing that the combination of a two character keyword, 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. 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 or 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_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-whitespace character of the last line of list, as in:

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_agruments(
    '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 3 disallows mixing the uses of 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 restriction, the line lengths should be limited to 72 characters.

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

Backslashes may still be appropriate at times. For example long, multiple with-statements cannot use implicit continuation, so backlashes are acceptable.

Should a line break before or after a binary operator?

For decades the recommended style was to break after binary operators. But this can hurt readability in two ways: the operators tend to get scattered across different columns on the screen, and each operator is moved away from its operand and onto the previous line. Here, the eye has to do extra work to tell which items are added and which are subtracted.

#No: Operators sit far away from their operands.

```
income = ( gross_wages +
           taxable_interest +
           (dividends - qualified_dividends) -
           ira_deduction -
           student_loan_interset)
```

To solve this readability problem, mathematicians & their publishers follow the opposite convention. Following the tradition from mathematics usually results in more readable code:

#Yes:

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest
```

In Python code, it is permissible to break before or after a binary operator, as long as the convention is consistent locally. For new code Knuth's style is suggested.

Blank Lines

Surround top-level function and class definition with two blank lines. Method definition inside a class are surrounded by a single blank line.
Extra blank lines may be used to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners.
Use blank lines in functions, sparingly, to indicate logical sections.

Source File Encoding:

Code in the core Python distribution should always use UTF-8.
For Python 3.0 and beyond, the following policy is prescribed for the standard library. All identifiers in the Python standard library must use ASCII-only identifiers; and should use English words wherever feasible. In addition, string literals and comments must also be in ASCII. The only exception are (a) test cases testing the non ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names.
Open source projects with a global audience are encouraged to adopt a similar policy.

Imports:

Imports should usually be on separate lines, eg:
Yes:

```
import os
import sys
```

No:

```
import os, sys
```

It's okay to say this though:

```
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 & 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 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 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.

Implicit relative imports should never be used and have been removed in Python 3.

Module level dunder names:
Module level "dunders" (i.e. names with two leading & two trailing underscore) such as __all__, __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.

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.

Whitespace in Expressions & Statements
Avoid trailing whitespace. Because it's usually invisible, it can be confusing. eg. a backslash followed by a space and a newline does not count as a line continuation marker. Some editors don't preserve it and many projects have pre-commit hooks that reject it.

Avoid extraneous whitespace in the following situations:
Immediately inside parentheses, brackets or braces.
Yes:

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

No:

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

Between a trailing comma & a following close parenthesis.

Yes:

```
foo = (0,)
```

No:

```
bar = (0, )
```

Immediately before a comma, semicolon, or colon:

Yes:

```
if x==4: print x,y; y=y,x
```

No:

```
if x==4 : print x , y ; x , y = y , x
```

Comments

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

Comment should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter.

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentences should end in a period.

Block Comments:

Block comments generally apply to some code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # & a single space.

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments:
Use inline comments sparingly.
An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.
Inline comments are unnecessary and in fact distracting if they state the obvious.
Don't do this:

```
x = x + 1                 # increment x
```

But something, this is useful.

```
x = x + 1                 # compensate for border
```

Documentation Strings:
Convention for writing good documentation strings are immortalized
Write docstrings for all public modules, functions, classes & methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the def line.

PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself.
Eg.

```
"""Return a foobang
    Optional plotz says to frobnicate the bizbaz first
    """
```

Naming Conventions:

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent - nevertheless, here are the currently recommended naming standards. New modules & packages should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Overriding Principle

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

Descreptive : 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:

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lowercase_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizeWords (or CapWords, CamelCase, StudlyCaps)
- mixedCase (differs from CapitalizedWords by initial lowercase character!)
- Capitalized_Words_With_Underscores (ugly!)

Note: When using abbreviations in CapWords, capitalize all the letters of the abbrevicdion.

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 indistiguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

ASCII Compatibility:
Identifiers used in the standard library must be ASCII compatible as described in the policy section of PEP.

Package and Modules Names:
Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

Class Names:
Class names should normally use the CapWords convention. The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.
Note that there is a separate convention for builtin names: most builtin names are single words, with the CapWords convention used only for exception names and builtin constants.

Type variable names:
Names of type variables introduced in PEP should normally use CapWords preferring short names: T, AnyStr, Num. It is recommended to add suffixes -co or -contra to the variables used to declare covariant or contravariant behaviour correspondingly.
Examples:

```
from typing import TypeVar
VT_co = TypeVar('VT_co', covariant=True)
KT_contra = TypeVar('KT_contra', contravariant=True)
```

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

Global Variable Names:
The conventions are about the same as those for functions. Modules that are designed for use via from M import * should use the __all__ mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore.

Function Names:
Function names should be lowercase, with words separated by underscores as necessary to improve readability. mixedCase is allowed only in contexts where that's already the prevailing style to retain backwards compatibility.

Function and method arguments:
Always use self for the first argument to instance methods. Always use cls for the first argument to class methods. If a function argument's name clashes with a reversed 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.

Methods Names and Instance Variables:
Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.
Use one leading underscore only for non-public methods & instance variables.
To avoid name clashes with subclasses, use two leading underscores to invoke Python'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. Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Constants:
Constants are usually defined on a module level & written in all capital letters with underscores seperating words. Examples include MAX_OVERFLOW & TOTAL.

Designing for inheritance:

Always decide whether a class's methods and instance variables should be public or non-public. If in dobt, chose non-public; it's easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no gurantees that non-public attributes won't change or even be removed.

We don't use the term "private" here, since no attribute is really private in Python.
Another category of attributes are those that are part of the "subclass API". Some classes are designed to be inherit from, either to extend or modify aspects of the class's behaviour. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

Public attributes should have no leading underscores.

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: Properties only work on new-style classes.
Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.
Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is cheap.

If your class is intended to be subclassed, & you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note2: Name mangling can make certain uses such as debugging and __getattr__(), less convenient. However the name mangling algorithm is well documented and easy to perform manually.

Note3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

Public & internal interfaces:

Any backwards compatibility gurantees apply only to public interfaces. Accordingly, it is important that users be able to clearly distinguish between public & 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 guarntees. 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 should still be prefixed with a single leading underscore. An interface is also considered if any containing namespace is considered internal.

Imported names should always be considered an implementation detail. Other modules must not rely on indirect access to such imported names unless they are an explicitly documented part of the containing module's API, such as os.path or a package's __init__ module that exposes functionality from submodules.

Programming Recommendation:
Code should be written in a way that does not disadvantage other implementations of Python.
For example, do not rely on CPython's efficient implementation of inplace string concatenation for statements in the form a+=b or a=a+b. This optimization is fragile even in CPython and isn't present at all in implementations that don't use refcounting. In performance sensitive parts of the library, the ''.join() form should be used instead. This will ensure that concatenation occurs in linear time across various implementation.

Comparisons to singletons like None should always be done with is or is not, never the equality operators.

Also, beware of writing if x when you really mean if x is not None eg. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type that could be false in a boolean context.

Use is not operator rather than not ... is. While both expressions are functionally identical, the former is more readable & preferred.

Yes:

```
if foo is not None:
```

No:

```
if not foo is None:
```

When implementing ordering operations which rich comparisons, it is best to implement all six operations rather than relying on other code to only exercise a particular comparison.

To minimize the effort involved, the functools.total_ordering() decorator provides a tool to generate missing comparison methods.

PEP 207 indicates that reflexivity rules are assumed by Python. Thus, the interpreter may swap y > x with x < y, y >= x with x <= y, and may swap the arguments of x == y & x != y. The sort() and min() operations are guaranteed to use the < operator and the max() function uses the

> operator. However, it is best to implements all six operations so that confusion doesn't arise in other contexts.

Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

Yes:

```
def f(x): return 2*x
```

No:

```
f = lambda x: 2*x
```

The first form means that the name of the resulting function object is specifically 'f' instead of the generic '<lambda>'. This is more useful for tracebacks & string representation in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement.

Derive exceptions from Exception rather than BaseException. Direct inheritance from BaseException is reserved for exceptions where catching them is almost always the wrong thing to do.

Design exceptions ~~from~~ hierarchies based on the distinctions that code catching the exceptions is likely to need, rather than the locations where the exceptions are raised. Aim to answer the question "What went wrong?" programmatically, rather than the locations where the exceptions are raised.

Class naming conventions apply here, although you should add the suffix "Error" to your exception classes if the exception is an error. Nonerror exceptions that are used for non-local flow control or others forms of signaling need no special suffix.

Use exception chaining appropriately. In Python 3, "raise X from Y" should be used to indicate explicit replacement without losing the original traceback.

When deliberately replacing an inner exception ensure that relevant details are transferred to the new exception.

When raising an exception in Python 2, use raise ValueError('message') instead of the older form raise ValueError, 'message'.

The latter form is not legal Python 3 syntax. The paren-using form also means that when the exception arguments are long or include string formatting, you don't need to use line continuation character thanks to the containing parentheses.

When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause.

For example, use:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

A bare except: clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C & can disguise other problems. If you want to catch all exceptions that signal program errors, use except Exception:

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

1. If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occured.
2. If the code needs to do some cleanup work, but then lets the exception propagate upwards with raise. try... finally can be better way to handle this case.

When binding caught exceptions to a name, prefer the explicit name binding syntax added in Python

```
try:
    process_data()
except Exception as exc:
    raise DataProcessFailedError(str(exc))
```

This is the only syntax supported in Python 3, & avoids the ambiguity problems associated with the older comma-based syntax.

When catching operating system errors, prefer the explicit exception hierarchy introduced in Python 33 over introspection of errno values.

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

Yes:

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

No:

```
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value
    return key_not_found(key)
```

When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly & reliably after use. A try/finally statement is also acceptable.
Context managers should be invoked through separate functions or methods whenever they do something other than acquire & release resource.
For example:

Yes:

```
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
```

No:

```
with conn:
    do_stuff_in_transaction(conn)
```

The latter example doesn't provide any information to indicate that the __enter__ & __exit__ methods are doing something other than closing the connection after a transaction. Being explicit is important in this case.

Be consistent in return statements. Either all return statements in a function should return an expression or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function.

Yes:

```
def foo(x):
    if x>=0:
        return math.sqrt(x)
    else:
        return None
def bar(x):
    if x<0:
        return None
    return math.sqrt(x)
```

No:

```
def foo(x):
    if x>=0:
        return math.sqrt(x)
def bar(x):
    if x<0:
        return
    return math.sqrt(x)
```

Using string methods instead of the string module.

String methods are always much faster & share the same API with unicode strings. Override this rule if backward compatibility with Python older than 2.0 or suffixes.

startswith() and endswith() are cleaner & less error prone.
For example:
Yes:

```
if foo.startswith('bar'):
```

No:

```
if foo[:3] == 'bar':
```

Object type comparisons should always use isinstance() instead of comparing types directly:

Yes:

```
if isinstance(obj, int):
```

No:

```
if type(obj) is type(1):
```

When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2, str & unicode have a common base class, basestring, so you can do:

```
if isinstance(obj, basestring):
```

For sequences, (strings, lists, tuples), use the fact that empty sequences are false:

Yes:

```
if not seq:
if seq:
```

No:

```
if len(seq):
if not len(seq):
```

Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable & some editors will trim them.

Function Annotations:

With the acceptance of PEP 484, the style rules for function annotations are changing.

In order to be forward compatible, function annotations in Python 3 code should preferably 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 easy it was to add those annotations, and observing whether 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 refactorings.

For code that wants to make a different use of function annotations it is recommended to put a comment of the form:

```
# type: ignore
```

near the top of the file; this tells type checker to ignore all annotations.