PEP 8 – Style Guide for Python Code

Author: Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Alyssa Coghlan <ncoghlan at gmail.com>

Status: Active

Type: Process

Created: 05-Jul-2001

ost-History: 05-Jul-2001, 01-Aug-2013

ıtroduction

ıis document gives coding conventions for the Python code comprising the standard librar the main Python distribution. Please see the companion informational PEP describing styl ıidelines for the C code in the C implementation of Python.

ıis document and PEP 257 (Docstring Conventions) were adapted from Guido's original ython Style Guide essay, with some additions from Barry's style guide [2].

ıis style guide evolves over time as additional conventions are identified and past ɔnventions are rendered obsolete by changes in the language itself.

any projects have their own coding style guidelines. In the event of any conflicts, such oject-specific guides take precedence for that project.

Foolish Consistency is the Hobgoblin of Little Minds

ne of Guido's key insights is that code is read much more often than it is written. The ıidelines provided here are intended to improve the readability of code and make it ɔnsistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".

style guide is about consistency. Consistency with this style guide is important. Consistenc ithin a project is more important. Consistency within one module or function is the most ıportant.

owever, know when to be inconsistent – sometimes style guide recommendations just arer ɔplicable. When in doubt, use your best judgment. Look at other examples and decide wha oks best. And don't hesitate to ask!

particular: do not break backwards compatibility just to comply with this PEP!

ɔme other good reasons to ignore a particular guideline:

1. When applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.
2. To be consistent with surrounding code that also breaks it (maybe for historic reasons) although this is also an opportunity to clean up someone else's mess (in true XP style).
3. Because the code in question predates the introduction of the guideline and there is n other reason to be modifying that code.
4. When the code needs to remain compatible with older versions of Python that don't support the feature recommended by the style guide.

ode Lay-out

dentation

se 4 spaces per indentation level.

ontinuation lines should align wrapped elements either vertically using Python's implicit lin ining inside parentheses, brackets and braces, or using a *hanging indent* [1]. When using a anging indent the following should be considered; there should be no arguments on the st line and further indentation should be used to clearly distinguish itself as a continuation ne:

```
# 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 re
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)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

ıe 4-space rule is optional for continuation lines.

ptional:

```
# Hanging indents *may* be indented to other than 4 spaces.
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

'hen the conditional part of an `if`-statement is long enough to require that it be written :ross multiple lines, it's worth noting that the combination of a two character keyword (i.e. `:`), plus a single space, plus an opening parenthesis creates a natural 4-space indent for the ıbsequent lines of the multiline conditional. This can produce a visual conflict with the dented suite of code nested inside the `if`-statement, which would also naturally be dented to 4 spaces. This PEP takes no explicit position on how (or whether) to further ;ually distinguish such conditional lines from the nested suite inside the `if`-statement. :ceptable 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()
```

ılso see the discussion of whether to break before or after binary operators below.)

ıe closing brace/bracket/parenthesis on multiline constructs may either line up under the
st 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',
    )
```

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

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

ıbs or Spaces?

ɔaces are the preferred indentation method.

bs should be used solely to remain consistent with code that is already indented with tabs

ython disallows mixing tabs and spaces for indentation.

laximum Line Length

mit all lines to a maximum of 79 characters.

ɔr flowing long blocks of text with fewer structural restrictions (docstrings or comments), th
ıe length should be limited to 72 characters.

miting the required editor window width makes it possible to have several files open side b
de, and works well when using code review tools that present the two versions in adjacent
ɔlumns.

ıe default wrapping in most tools disrupts the visual structure of the code, making it more
fficult to understand. The limits are chosen to avoid wrapping in editors with the window
idth set to 80, even if the tool places a marker glyph in the final column when wrapping

ıes. Some web based tools may not offer dynamic line wrapping at all.

ɔme teams strongly prefer a longer line length. For code maintained exclusively or primarily
/ a team that can reach agreement on this issue, it is okay to increase the line length limit
ɔ to 99 characters, provided that comments and docstrings are still wrapped at 72
ıaracters.

ıe Python standard library is conservative and requires limiting lines to 79 characters (and
ɔcstrings/comments to 72).

ıe preferred way of wrapping long lines is by using Python's implied line continuation insid
ırentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping
:pressions in parentheses. These should be used in preference to using a backslash for line
ɔntinuation.

ıckslashes may still be appropriate at times. For example, long, multiple `with`-statements
ɔuld not use implicit continuation before Python 3.10, so backslashes were acceptable for
ıat case:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

ee the previous discussion on multiline if-statements for further thoughts on the
dentation of such multiline `with`-statements.)

ıother such case is with `assert` statements.

ake sure to indent the continued line appropriately.

ıould a Line Break Before or After a Binary Operator?

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

```
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

› solve this readability problem, mathematicians and their publishers follow the opposite ›nvention. Donald Knuth explains the traditional rule in his *Computers and Typesetting* ›ries: "Although formulas within a paragraph always break after binary operations and ›lations, displayed formulas always break before binary operations" [3].

›llowing the tradition from mathematics usually results in more readable code:

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

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

ank Lines

ırround top-level function and class definitions with two blank lines.

ethod definitions inside a class are surrounded by a single blank line.

‹tra blank lines may be used (sparingly) to separate groups of related functions. Blank lines ay be omitted between a bunch of related one-liners (e.g. a set of dummy ıplementations).

›e blank lines in functions, sparingly, to indicate logical sections.

/thon accepts the control-L (i.e. ^L) form feed character as whitespace; many tools treat ese characters as page separators, so you may use them to separate pages of related ‹ctions of your file. Note, some editors and web-based code viewers may not recognize ›ntrol-L as a form feed and will show another glyph in its place.

›urce File Encoding

›de in the core Python distribution should always use UTF-8, and should not have an ıcoding declaration.

the standard library, non-UTF-8 encodings should be used only for test purposes. Use nor SCII characters sparingly, preferably only to denote places and human names. If using non-SCII characters as data, avoid noisy Unicode characters like zalgo and byte order marks.

l identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD ;e English words wherever feasible (in many cases, abbreviations and technical terms are ;ed which aren't English).

pen source projects with a global audience are encouraged to adopt a similar policy.

ıports

- 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. 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 (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on `sys.path`

```
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.

- 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`.

- Wildcard imports (`from <module> import *`) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools. There is one defensible use case for a wildcard import, which is to republish an internal interface as part of a public API (for example, overwriting a pure Python implementation of an interface with the definitions from an optional accelerato module and exactly which definitions will be overwritten isn't known in advance).

  When republishing names this way, the guidelines below regarding public and internal interfaces still apply.

odule Level Dunder Names

odule level "dunders" (i.e. names with two leading and two trailing underscores) such as `_all__`, `__author__`, `__version__`, etc. should be placed after the module docstring but before ıy import statements *except* `from __future__` imports. Python mandates that future-imports ust 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
```

tring Quotes

Python, single-quoted strings and double-quoted strings are the same. This PEP does not ake a recommendation for this. Pick a rule and stick to it. When a string contains single or ouble quote characters, however, use the other one to avoid backslashes in the string. It nproves readability.

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

/hitespace in Expressions and Statements

et Peeves

/oid extraneous whitespace in the following situations:

- Immediately inside parentheses, 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:

```
# 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 amoun on either side (treating it as the operator with the lowest priority). 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[: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 : : step]
ham[ : upper]
```

- Immediately before the open parenthesis that starts the argument list of a function cal

```
# 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 anothe

```
# Correct:
x = 1
y = 2
long_variable = 3
```

```
# Wrong:
x             = 1
y             = 2
long_variable = 3
```

ther Recommendations

- Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuatic marker. Some editors don't preserve it and many projects (like CPython itself) have pre commit hooks that reject it.

- Always surround these binary operators with a single space on either side: assignment (`=`), augmented assignment (`+=`, `-=` etc.), comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), Booleans (`and`, `or`, `not`).

- If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides o a binary operator:

```
# Correct:
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

```
# Wrong:
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- Function annotations should use the normal rules for colons and always have spaces around the `->` arrow if present. (See Function Annotations below for more about function annotations.):

```
# Correct:
def munge(input: AnyStr): ...
def munge() -> PosInt: ...
```

```
# Wrong:
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

- Don't use spaces around the `=` sign when used to indicate a keyword argument, or when used to indicate a default value for an *unannotated* function parameter:

```
# Correct:
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

```
# Wrong:
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

When combining an argument annotation with a default value, however, do use spaces around the `=` sign:

```
# Correct:
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...
```

```
# Wrong:
def munge(input: AnyStr=None): ...
def munge(input: AnyStr, limit = 1000): ...
```

- Compound statements (multiple statements on the same line) are generally discouraged:

```
# Correct:
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

/hen to Use Trailing Commas

ailing commas are usually optional, except they are mandatory when making a tuple of on ement. For clarity, it is recommended to surround the latter in (technically redundant) arentheses:

```
# Correct:
FILES = ('setup.cfg',)
```

```
# Wrong:
FILES = 'setup.cfg',
```

'hen trailing commas are redundant, they are often helpful when a version control system i ed, when a list of values, arguments or imported items is expected to be extended over ne. The pattern is to put each value (etc.) on a line by itself, always adding a trailing comm d add the close parenthesis/bracket/brace on the next line. However it does not make nse to have a trailing comma on the same line as the closing delimiter (except in the abov se of singleton tuples):

```
# Correct:
FILES = [
    'setup.cfg',
    'tox.ini',
    ]
initialize(FILES,
           error=True,
           )
```

```
# Wrong:
FILES = ['setup.cfg', 'tox.ini',]
initialize(FILES, error=True,)
```

omments

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

omments should be complete sentences. The first word should be capitalized, unless it is a entifier that begins with a lower case letter (never alter the case of identifiers!).

ock comments generally consist of one or more paragraphs built out of complete ntences, with each sentence ending in a period.

ou should use one or two spaces after a sentence-ending period in multi-sentence omments, except after the final sentence.

nsure that your comments are clear and easily understandable to other speakers of the nguage you are writing in.

ython coders from non-English speaking countries: please write your comments in English, nless you are 120% sure that the code will never be read by people who don't speak your nguage.

ock Comments

ock comments generally apply to some (or all) code that follows them, and are indented tc e same level as that code. Each line of a block comment starts with a `#` and a single space nless it is indented text inside the comment).

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

line Comments

se inline comments sparingly.

n inline comment is a comment on the same line as a statement. Inline comments should b parated by at least two spaces from the statement. They should start with a # and a single ace.

line comments are unnecessary and in fact distracting if they state the obvious. Don't do is:

```
x = x + 1                 # Increment x
```

ut sometimes, this is useful:

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

ocumentation Strings

onventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in EP 257.

- Write docstrings for all public modules, functions, classes, and 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:

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

- For one liner docstrings, please keep the closing `"""` on the same line:

```
"""Return an ex-parrot."""
```

aming Conventions

ıe naming conventions of Python's library are a bit of a mess, so we'll never get this ɔmpletely consistent – nevertheless, here are the currently recommended naming standard ew modules and packages (including third party frameworks) should be written to these andards, but where an existing library has a different style, internal consistency is preferred

verriding Principle

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

escriptive: Naming Styles

ıere are a lot of different naming styles. It helps to be able to recognize what naming style ɛing used, independently from what they are used for.

ıe following naming styles are commonly distinguished:

- `b` (single lowercase letter)

- `B` (single uppercase letter)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords, or CamelCase – so named because of the bumpy look of its letters [4]). This is also sometimes known as StudlyCaps.

  Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thu HTTPServerError is better than HttpServerError.
- `mixedCase` (differs from CapitalizedWords by initial lowercase character!)
- `Capitalized_Words_With_Underscores` (ugly!)

ıere's also the style of using a short unique prefix to group related names together. This is ɔt used much in Python, but it is mentioned for completeness. For example, the `os.stat()` nction returns a tuple whose items traditionally have names like `st_mode`, `st_size`, `st_mtime` ıd so on. (This is done to emphasize the correspondence with the fields of the POSIX syste ıll struct, which helps programmers familiar with that.)

ıe X11 library uses a leading X for all its public functions. In Python, this style is generally ɛemed unnecessary because attribute and method names are prefixed with an object, and nction names are prefixed with a module name.

addition, the following special forms using leading or trailing underscores are recognized ıese can generally be combined with any case convention):

- `_single_leading_underscore`: weak "internal use" indicator. E.g. `from M import *` does not import objects whose names start with an underscore.
- `single_trailing_underscore_`: used by convention to avoid conflicts with Python keywor e.g. :

  ```
  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).

- `__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.

escriptive: Naming Conventions

ames to Avoid

ever use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase tter eye) as single character variable names.

some fonts, these characters are indistinguishable from the numerals one and zero. When mpted to use ‘l’, use ‘L’ instead.

CII Compatibility

entifiers used in the standard library must be ASCII compatible as described in the policy ction of PEP 3131.

ckage and Module Names

odules should have short, all-lowercase names. Underscores can be used in the module ame if it improves readability. Python packages should also have short, all-lowercase name though the use of underscores is discouraged.

hen an extension module written in C or C++ has an accompanying Python module that ovides a higher level (e.g. more object oriented) interface, the C/C++ module has a leadin nderscore (e.g. `_socket`).

ass Names

ass names should normally use the CapWords convention.

ne naming convention for functions may be used instead in cases where the interface is ocumented and used primarily as a callable.

ote that there is a separate convention for builtin names: most builtin names are single ords (or two words run together), with the CapWords convention used only for exception ames and builtin constants.

pe Variable Names

ames of type variables introduced in PEP 484 should normally use CapWords preferring ıort names: `T`, `AnyStr`, `Num`. It is recommended to add suffixes `_co` or `_contra` to the variable ;ed to declare covariant or contravariant behavior correspondingly:

```
from typing import TypeVar

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

ception Names

ecause exceptions should be classes, the class naming convention applies here. However, ou should use the suffix "Error" on your exception names (if the exception actually is an ror).

obal Variable Names

et's hope that these variables are meant for use inside one module only.) The conventions e about the same as those for functions.

odules that are designed for use via `from M import *` should use the `__all__` mechanism to event exporting globals, or use the older convention of prefixing such globals with an nderscore (which you might want to do to indicate these globals are "module non-public").

nction and Variable Names

ınction names should be lowercase, with words separated by underscores as necessary to ıprove readability.

ariable names follow the same convention as function names.

ixedCase is allowed only in contexts where that's already the prevailing style (e.g. reading.py), to retain backwards compatibility.

nction and Method Arguments

ways use `self` for the first argument to instance methods.

ways use `cls` for the first argument to class methods.

a function argument's name clashes with a reserved keyword, it is generally better to ıpend a single trailing underscore rather than use an abbreviation or spelling corruption. ıus `class_` is better than `clss`. (Perhaps better is to avoid such clashes by using a synonym

ethod Names and Instance Variables

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

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

avoid name clashes with subclasses, use two leading underscores to invoke Python's nam
angling rules.

ython mangles these names with the class name: if class Foo has an attribute named `__a`, it
nnot be accessed by `Foo.__a`. (An insistent user could still gain access by calling
o._Foo__a`.) Generally, double leading underscores should be used only to avoid name
nflicts with attributes in classes designed to be subclassed.

ote: there is some controversy about the use of __names (see below).

nstants

onstants are usually defined on a module level and written in all capital letters with
nderscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.

esigning for Inheritance

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

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

e don't use the term "private" here, since no attribute is really private in Python (without a
enerally unnecessary amount of work).

nother category of attributes are those that are part of the "subclass API" (often called
rotected" in other languages). Some classes are designed to be inherited from, either to
tend or modify aspects of the class's behavior. When designing such a class, take care to
ake explicit decisions about which attributes are public, which are part of the subclass API,
nd which are truly only to be used by your base class.

ith 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. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

  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 gro 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.

- If your class is intended to be subclassed, and you have attributes that you do not wan 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.

  Note 1: 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.

  Note 2: Name mangling can make certain uses, such as debugging and `__getattr__()`, less convenient. However the name mangling algorithm is well documented and easy t perform manually.

  Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidenta name clashes with potential use by advanced callers.

ublic and Internal Interfaces

ny backwards compatibility guarantees apply only to public interfaces. Accordingly, it is nportant that users be able to clearly distinguish between public and internal interfaces.

ocumented interfaces are considered public, unless the documentation explicitly declares em to be provisional or internal interfaces exempt from the usual backwards compatibility uarantees. All undocumented interfaces should be assumed to be internal.

o better support introspection, modules should explicitly declare the names in their public PI using the `__all__` attribute. Setting `__all__` to an empty list indicates that the module ha o public API.

ven with `__all__` set appropriately, internal interfaces (packages, modules, classes, function tributes or other names) should still be prefixed with a single leading underscore.

n interface is also considered internal if any containing namespace (package, module or ass) is considered internal.

mported names should always be considered an implementation detail. Other modules mus ot rely on indirect access to such imported names unless they are an explicitly documented art of the containing module's API, such as `os.path` or a package's `__init__` module that xposes functionality from submodules.

rogramming Recommendations

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

  For example, do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form `a += b` or `a = a + b`. This optimization is fragile even in CPython (it only works for some types) 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 implementations.

- 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` – e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) 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 and preferred:

```
# Correct:
if foo is not None:
```

```
# Wrong:
if not foo is None:
```

- When implementing ordering operations with rich comparisons, it is best to implemen all six operations (`__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, `__ge__`) 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 ma swap `y > x` with `x < y`, `y >= x` with `x <= y`, and may swap the arguments of `x == y` and `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 implement 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:

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

  ```
  # Wrong:
  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 and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

- 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 exception 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 only

stating that "A problem occurred" (see PEP 3151 for an example of this lesson being learned for the builtin exception hierarchy)

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

- Use exception chaining appropriately. `raise X from Y` should be used to indicate explic replacement without losing the original traceback.

  When deliberately replacing an inner exception (using `raise X from None`), ensure that relevant details are transferred to the new exception (such as preserving the attribute name when converting KeyError to AttributeError, or embedding the text of the origina exception in the new exception message).

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

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

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

  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 occurred.
  2. If the code needs to do some cleanup work, but then lets the exception propagat upwards with `raise`. `try...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 `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:

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

```
# Wrong:
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 and 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 and release resources:

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

  ```
  # Wrong:
  with conn:
      do_stuff_in_transaction(conn)
  ```

  The latter example doesn't provide any information to indicate that the `__enter__` and `__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 (if reachable):

```
# Correct:

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)
```

```
# Wrong:

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

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

- Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes.

  startswith() and endswith() are cleaner and less error prone:

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

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

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

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

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

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

```
# Correct:
if not seq:
if seq:
```

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

- Don't write string literals that rely on significant trailing 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:
```

- Use of the flow control statements `return`/`break`/`continue` within the finally suite of a `try...finally`, where the flow control statement would jump outside the finally suite, is discouraged. This is because such statements will implicitly cancel any active exception that is propagating through the finally suite:

```
# Wrong:
def foo():
    try:
        1 / 0
    finally:
        return 42
```

unction Annotations

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

- Function annotations should use PEP 484 syntax (there are some formatting recommendations for annotations in the previous section).

- 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 checkers to ignore all annotations. (More fine-grained ways of disabling complaints from type checkers can be found in PEP 484.)

- 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.
- 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 checkers 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. Stub files can be distributed with a library, or separately (with the library author's permission) through the typeshed repo [5].

ariable Annotations

EP 526 introduced variable annotations. The style recommendations for them are similar to ose on function annotations described above:

- Annotations for module level variables, class and instance 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 (see PEP 484 for details).

ootnotes

]

Hanging indentation is a type-setting style where all the lines in a paragraph are indented except the first line. In the context of Python, the term is used to describe a style where the opening parenthesis of a parenthesized statement is the last non-whitespace character of the line, with subsequent lines being indented until the closing parenthesis.

eferences

]

Barry's GNU Mailman style guide http://barry.warsaw.us/software/STYLEGUIDE.txt

]

Donald Knuth's *The TeXBook*, pages 195 and 196.

]

http://www.wikipedia.com/wiki/CamelCase

]

Typeshed repo https://github.com/python/typeshed

opyright

---

ource: https://github.com/python/peps/blob/main/peps/pep-0008.rst

ıst modified: 2023-12-09 16:19:37 GMT