PEP 8 — the Style Guide for Python Code

This stylized presentation of the well-established PEP 8 was created by Kenneth Reitz (for humans).

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python [1].

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

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

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

A Foolish Consistency is the Hobgoblin of Little Minds

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

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

However, know when to be inconsistent—sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

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

Some 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 no 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.

Code lay-out

Indentation

Use 4 spaces per indentation 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* [3]. 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.

Yes:

```
# Aligned with opening delimiter.
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)
```

No:

```
# Arguments on first line forbidden when not using vertical al
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

```
# Further indentation required as indentation is not distingui
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 noting that the combination of a two character keyword (i.e. **`if`**), 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 (or whether) 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 editor
# 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 l
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

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

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_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

or it may be lined up under the first character of the line that starts the multi-line construct, as in:

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

Tabs or Spaces?

Spaces are the preferred indentation method.

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

Python 3 disallows mixing the use of tabs and spaces for indentation.

Python 2 code indented with a mixture of tabs and spaces should be converted to using spaces exclusively.

When invoking the Python 2 command line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When using `-tt` these warnings become errors. These options are highly recommended!

Maximum Line Length

Limit all lines to a maximum of 79 characters.

For flowing long blocks of text with fewer structural restrictions (docstrings or comments), 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 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.

Some teams strongly prefer a longer line length. For code maintained exclusively or primarily by a team that can reach agreement on this issue, it is okay to increase the nominal line length from 80 to 100 characters (effectively increasing the maximum length to 99 characters), provided that comments and docstrings are still wrapped at 72 characters.

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

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

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

```
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())
```

(See the previous discussion on multiline if-statements for further thoughts on the indentation of such multiline **`with`**-statements.)

Another such case is with **`assert`** statements.
Make sure to indent the continued line appropriately.

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_interest)
```

To solve this readability problem, mathematicians and their publishers follow the opposite convention. Donald Knuth explains the traditional rule in his *Computers and Typesetting* series:

> "Although formulas within a paragraph always break after binary operations and relations, displayed formulas always break before binary operations" [4].

Following the tradition from mathematics usually results in more readable code:

```
# Yes: easy to match operators with operands
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 definitions with two blank lines.

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

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

Use blank lines in functions, sparingly, to indicate logical sections.

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

Source File Encoding

Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).

Files using ASCII (in Python 2) or UTF-8 (in Python 3) should not have an encoding declaration.

In the standard library, non-default encodings should be used only for test purposes or when a comment or docstring needs to mention an author name that contains non-ASCII characters; otherwise, using `\x`, `\u`, `\U`, or `\N` escapes is the preferred way to include non-ASCII data in string literals.

For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions 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, e.g.:
>
> 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 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.

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

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 :

```
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 accelerator 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.

Module level dunder names

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

For example:

```
"""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
```

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:

Yes:

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

No:

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

Between a trailing comma and a following close parenthesis:

Yes:

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

No:

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

Immediately before a comma, semicolon, or colon:

Yes:

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

No:

```
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 (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.

Yes:

```
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]
```

No:

```
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:

Yes:

```
spam(1)
```

No:

```
spam (1)
```

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

Yes:

```
dct['key'] = lst[index]
```

No:

```
dct ['key'] = lst [index]
```

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

Yes:

```
x = 1
y = 2
long_variable = 3
```

No:

```
x             = 1
y             = 2
long_variable = 3
```

Other 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 continuation 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 of a binary operator.

Yes:

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

No:

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

Don't use spaces around the `=` sign when used to indicate a keyword argument or a default parameter value.

Yes:

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

No:

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

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.)

Yes:

```
def munge(input: AnyStr): ...
def munge() -> AnyStr: ...
```

No:

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

When combining an argument annotation with a default value, use spaces around the `=` sign (but only for those arguments that have both an annotation and a default).

Yes:

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

No:

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

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

Yes:

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

Rather not:

```
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:

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

Definitely not:

```
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()
```

When to use trailing commas

Trailing commas are usually optional, except they are mandatory when making a tuple of one element (and in Python 2 they have semantics for the **`print`** statement). For clarity, it is recommended to surround the latter in (technically redundant) parentheses.

Yes:

```
FILES = ('setup.cfg',)
```

OK, but confusing:

```
FILES = 'setup.cfg',
```

When trailing commas are redundant, they are often helpful when a version control system is used, when a list of values, arguments or imported items is expected to be extended over time. The pattern is to put each value (etc.) on a line by itself, always adding a trailing comma, and add the close parenthesis/bracket/brace on the next line. However it does not make sense to have a trailing comma on the same line as the closing delimiter (except in the above case of singleton tuples).

Yes:

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

No:

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

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!

Comments 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 (never alter the case of identifiers!).

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 sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, follow Strunk and White.

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

Block Comments

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

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 sometimes, this is useful:

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

Documentation Strings

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in [PEP 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, e.g.:
>
> ```
> """Return a foobang
>
> Optional plotz says to frobnicate the bizbaz first.
> """
> ```
>
> For one liner docstrings, please keep the closing `"""` on the same line.

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 and packages (including third party frameworks) 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.

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:

`b` (single lowercase letter)

`B` (single uppercase letter)

`lowercase`

`lower_case_with_underscores`

`UPPERCASE`

`UPPER_CASE_WITH_UNDERSCORES`

`CapitalizedWords` (or CapWords, CamelCase[5], 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 abbreviation. Thus `HTTPServerError` is better than `HttpServerError`.

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. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

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

In addition, the following special forms using leading or trailing underscores are recognized (these 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 name starts with an underscore.

`single_trailing_underscore_`: used by convention to avoid conflicts with Python keyword, 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.

Prescriptive: Naming Conventions

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.

ASCII Compatibility

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

Package and Module 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.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. **_socket**).

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 (or two words run together), with the CapWords convention used only for exception names and builtin constants.

Type variable names

Names of type variables introduced in PEP 484 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 behavior 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 (if the exception actually is an error).

Global Variable Names

(Let’s hope that these variables are meant for use inside one module only.) 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 (which you might want to do to indicate these globals are “module non-public”).

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 (e.g. threading.py), 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 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

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 and 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`. (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 (see below).

Constants

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` 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 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 guarantees 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 (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. 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. (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 grow functional behavior. 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 (relatively) cheap.

If your class is intended to be subclassed, and 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.

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 to perform manually.

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

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 interfaces 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 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 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.
>
> Yes:
>
> ```
> if foo is not None:
> ```
>
> No:
>
> ```
> if not foo is None:
> ```
>
> When implementing ordering operations with rich comparisons, it is best to implement 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 may 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.

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 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 your 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. 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 (using "raise X" in Python 2 or "raise X from None" in Python 3.3+), 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 original exception in the new exception message).

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 characters 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, 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 propagate upwards with **`raise`**. **`try...finally`** can be a better way to handle this case.

When binding caught exceptions to a name, prefer the explicit name binding syntax added in Python 2.6:

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

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

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.

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 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. 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__`** 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).

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)
```

Use string methods instead of the string module.

String methods are always much faster and share the same API with unicode strings. Override this rule if backward compatibility with Pythons older than 2.0 is required.

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

`startswith()` and **`endswith()`** are cleaner and 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 and unicode have a common base class, basestring, so you can do:

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

Note that in Python 3, **`unicode`** and **`basestring`** no longer exist (there is only **`str`**) and a bytes object is no longer a kind of string (it is a sequence of integers instead)

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 and some editors (or more recently, reindent.py) will trim them.
Don't compare boolean values to True or False using ==:

Yes:

```
if greeting:
```

No:

```
if greeting == True:
```

Worse:

```
if greeting is True:
```

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. (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 checker 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 [6].

For code that needs to be backwards compatible, type annotations can be added in the form of comments. See the relevant section of PEP 484 [7].

Footnotes

1. PEP 7, Style Guide for C Code, van Rossum↩

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

3. *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.↩

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

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

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

7. Suggested syntax for Python 2.7 and straddling code https://www.python.org/dev/peps/pep-0484/#suggested-syntax-for-python-2-7-and-straddling-code↩

Copyright