

Google Python Style Guide

Revision 2.59

Amit Patel
Antoine Picard
Eugene Jhong
Jeremy Hylton
Matt Smart
Mike Shields

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: . You may toggle all summaries with the big arrow button:



 Toggle all summaries

Table of Contents

Python Language Rules	Lint Imports Packages Exceptions Global variables Nested/Local/Inner Classes and Functions List Comprehensions Default Iterators and Operators Generators Lambda Functions Conditional Expressions Default Argument Values Properties True/False evaluations Deprecated Language Features Lexical Scoping Function and Method Decorators Threading Power Features
Python Style Rules	Semicolons Line length Parentheses Indentation Blank Lines Whitespace Shebang Line Comments Classes Strings Files and Sockets TODO Comments Imports formatting Statements Access Control Naming Main

Important Note

Displaying Hidden Details in this Guide

[link](#)  This style guide contains many details that are initially hidden from view. They are marked by the triangle icon, which you see here on your left. Click it now. You should see "Hooray" appear below.

Hooray! Now you know you can expand points to get more details. Alternatively, there's a "toggle all" at the top of this document.


Background

Python is the main scripting language used at Google. This style guide is a list of *dos* and *don'ts* for Python programs.

To help you format code correctly, we've created a [settings file for Vim](#). For Emacs, the default settings should be fine.

Python Language Rules

Lint

[link](#)  Run `pylint` over your code.

Definition:

pylint is a tool for finding bugs and style problems in Python source code. It finds problems that are typically caught by a compiler for less dynamic languages like C and C++. Because of the dynamic nature of Python, some warnings may be incorrect; however, spurious warnings should be fairly infrequent.

Pros:

Catches easy-to-miss errors like typos, using-vars-before-assignment, etc.

Cons:

`pylint` isn't perfect. To take advantage of it, we'll need to sometimes: a) Write around it b) Suppress its warnings or c) Improve it.

Decision:

Make sure you run `pylint` on your code. Suppress warnings if they are inappropriate so that other issues are not hidden.

To suppress warnings, you can set a line-level comment:

```
dict = 'something awful' # Bad Idea... pylint: disable=redefined-builtin
```

pylint warnings are each identified by a alphanumeric code (C0112) and a symbolic name (`empty-docstring`). Prefer the symbolic names in new code or when updating existing code.

If the reason for the suppression is not clear from the symbolic name, add an explanation.

Suppressing in this way has the advantage that we can easily search for suppressions and revisit them.

You can get a list of pylint warnings by doing `pylint --list-msgs`. To get more information on a particular message, use `pylint --help-msg=C6409`.

Prefer `pylint: disable` to the deprecated older form `pylint: disable-msg`.

Unused argument warnings can be suppressed by using ``_`` as the identifier for the unused argument or prefixing the argument name with ``unused_``. In situations where changing the argument names is infeasible, you can mention them at the beginning of the function. For example:

```
def foo(a, unused_b, unused_c, d=None, e=None):
    _ = d, e
    return a
```

Imports

[link](#)

☐ Use `imports` for packages and modules only.

Definition:

Reusability mechanism for sharing code from one module to another.

Pros:

The namespace management convention is simple. The source of each identifier is indicated in a consistent way; `x.Obj` says that object `Obj` is defined in module `x`.

Cons:

Module names can still collide. Some module names are inconveniently long.

Decision:

Use `import x` for importing packages and modules.

Use `from x import y` where `x` is the package prefix and `y` is the module name with no prefix.

Use `from x import y as z` if two modules named `y` are to be imported or if `y` is an inconveniently long name.

For example the module `sound.effects.echo` may be imported as follows:

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

Do not use relative names in imports. Even if the module is in the same package, use the full package name. This helps prevent unintentionally importing a package twice.

Packages

[link](#)

☐ Import each module using the full pathname location of the module.

Pros:

Avoids conflicts in module names. Makes it easier to find modules.

Cons:

Makes it harder to deploy code because you have to replicate the package hierarchy.

Decision:

All new code should import each module by its full package name.

Imports should be as follows:

```
# Reference in code with complete name.
import sound.effects.echo

# Reference in code with just module name (preferred).
from sound.effects import echo
```

Exceptions

[link](#)

☐ Exceptions are allowed but must be used carefully.

Definition:

Exceptions are a means of breaking out of the normal flow of control of a code block to handle errors or other exceptional conditions.

Pros:

The control flow of normal operation code is not cluttered by error-handling code. It also allows the control flow to skip multiple frames when a certain condition occurs, e.g., returning from `N` nested functions in one step instead of having to carry-through error codes.

Cons:

May cause the control flow to be confusing. Easy to miss error cases when making library calls.

Decision:

Exceptions must follow certain conditions:

- Raise exceptions like this: `raise MyException('Error message')` or `raise MyException`. Do not use the two-argument form (`raise MyException, 'Error message'`) or deprecated string-based exceptions (`raise 'Error message'`).
- Modules or packages should define their own domain-specific base exception class, which should inherit from the built-in `Exception` class. The base exception for a module should be called `Error`.

```
class Error(Exception):
    pass
```

- Never use catch-all `except:` statements, or catch `Exception` or `StandardError`, unless you are re-raising the exception or in the outermost block in your thread (and printing an error message). Python is very tolerant in this regard and `except:` will really catch everything including misspelled names, `sys.exit()` calls, `Ctrl+C` interrupts, unittest failures and all kinds of other exceptions that you simply don't want to catch.
- Minimize the amount of code in a `try/except` block. The larger the body of the `try`, the more likely that an exception will be raised by a line of code that you didn't expect to raise an exception. In those cases, the `try/except` block hides a real error.

- Use the `finally` clause to execute code whether or not an exception is raised in the `try` block. This is often useful for cleanup, i.e., closing a file.
- When capturing an exception, use `as` rather than a comma. For example:

```
try:
    raise Error
except Error as error:
    pass
```

Global variables

[link](#)
☐ Avoid global variables.

Definition:

Variables that are declared at the module level.

Pros:

Occasionally useful.

Cons:

Has the potential to change module behavior during the import, because assignments to module-level variables are done when the module is imported.

Decision:

Avoid global variables in favor of class variables. Some exceptions are:

- Default options for scripts.
- Module-level constants. For example: `PI = 3.14159`. Constants should be named using all caps with underscores; see [Naming](#) below.
- It is sometimes useful for globals to cache values needed or returned by functions.
- If needed, globals should be made internal to the module and accessed through public module level functions; see [Naming](#) below.

Nested/Local/Inner Classes and Functions

[link](#)
☐ Nested/local/inner classes and functions are fine.

Definition:

A class can be defined inside of a method, function, or class. A function can be defined inside a method or function. Nested functions have read-only access to variables defined in enclosing scopes.

Pros:

Allows definition of utility classes and functions that are only used inside of a very limited scope. Very [ADI-y](#).

Cons:

Instances of nested or local classes cannot be pickled.

Decision:

They are fine.

List Comprehensions

[link](#)
☐ Okay to use for simple cases.

Definition:

List comprehensions and generator expressions provide a concise and efficient way to create lists and iterators without resorting to the use of `map()`, `filter()`, or `lambda`.

Pros:

Simple list comprehensions can be clearer and simpler than other list creation techniques. Generator expressions can be very efficient, since they avoid the creation of a list entirely.

Cons:

Complicated list comprehensions or generator expressions can be hard to read.

Decision:

Okay to use for simple cases. Each portion must fit on one line: mapping expression, `for` clause, filter expression. Multiple `for` clauses or filter expressions are not permitted. Use loops instead when things get more complicated.

```
Yes:
result = []
for x in range(10):
    for y in range(5):
        if x * y > 10:
            result.append((x, y))

for x in xrange(5):
    for y in xrange(5):
        if x != y:
            for z in xrange(5):
                if y != z:
                    yield (x, y, z)

return ((x, complicated_transform(x))
        for x in long_generator_function(parameter)
        if x is not None)

squares = [x * x for x in range(10)]
```

```
eat(jelly_bean for jelly_bean in jelly_beans
    if jelly_bean.color == 'black')
```

No:

```
result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]

return ((x, y, z)
        for x in xrange(5)
        for y in xrange(5)
        if x != y
        for z in xrange(5)
        if y != z)
```

Default Iterators and Operators

[link](#)

☒ Use default iterators and operators for types that support them, like lists, dictionaries, and files.

Definition:

Container types, like dictionaries and lists, define default iterators and membership test operators ("in" and "not in").

Pros:

The default iterators and operators are simple and efficient. They express the operation directly, without extra method calls. A function that uses default operators is generic. It can be used with any type that supports the operation.

Cons:

You can't tell the type of objects by reading the method names (e.g. `has_key()` means a dictionary). This is also an advantage.

Decision:

Use default iterators and operators for types that support them, like lists, dictionaries, and files. The built-in types define iterator methods, too. Prefer these methods to methods that return lists, except that you should not mutate a container while iterating over it.

Yes:

```
for key in adict: ...
if key not in adict: ...
if obj in alist: ...
for line in afile: ...
for k, v in dict.iteritems(): ...
```

No:

```
for key in adict.keys(): ...
if not adict.has_key(key): ...
for line in afile.readlines(): ...
```

Generators

[link](#)

☒ Use generators as needed.

Definition:

A generator function returns an iterator that yields a value each time it executes a `yield` statement. After it yields a value, the runtime state of the generator function is suspended until the next value is needed.

Pros:

Simpler code, because the state of local variables and control flow are preserved for each call. A generator uses less memory than a function that creates an entire list of values at once.

Cons:

None.

Decision:

Fine. Use "Yields:" rather than "Returns:" in the doc string for generator functions.

Lambda Functions

[link](#)

☒ Okay for one-liners.

Definition:

Lambdas define anonymous functions in an expression, as opposed to a statement. They are often used to define callbacks or operators for higher-order functions like `map()` and `filter()`.

Pros:

Convenient.

Cons:

Harder to read and debug than local functions. The lack of names means stack traces are more difficult to understand. Expressiveness is limited because the function may only contain an expression.

Decision:

Okay to use them for one-liners. If the code inside the lambda function is any longer than 60–80 chars, it's probably better to define it as a regular (nested) function.

For common operations like multiplication, use the functions from the `operator` module instead of lambda functions. For example, prefer `operator.mul` to `lambda x, y: x * y`.

Conditional Expressions

[link](#)

☒ Okay for one-liners.

Definition:

Conditional expressions are mechanisms that provide a shorter syntax for if statements. For example: `x = 1 if cond else 2`.

Pros:

Shorter and more convenient than an if statement.

Cons:

May be harder to read than an if statement. The condition may be difficult to locate if the expression is long.

Decision:

Okay to use for one-liners. In other cases prefer to use a complete if statement.

Default Argument Values
[link](#)
☐ Okay in most cases.
Definition:

You can specify values for variables at the end of a function's parameter list, e.g., `def foo(a, b=0):`. If `foo` is called with only one argument, `b` is set to 0. If it is called with two arguments, `b` has the value of the second argument.

Pros:

Often you have a function that uses lots of default values, but—rarely—you want to override the defaults. Default argument values provide an easy way to do this, without having to define lots of functions for the rare exceptions. Also, Python does not support overloaded methods/functions and default arguments are an easy way of “faking” the overloading behavior.

Cons:

Default arguments are evaluated once at module load time. This may cause problems if the argument is a mutable object such as a list or a dictionary. If the function modifies the object (e.g., by appending an item to a list), the default value is modified.

Decision:

Okay to use with the following caveat:

Do not use mutable objects as default values in the function or method definition.

```
Yes: def foo(a, b=None):
    if b is None:
        b = []
```

```
No: def foo(a, b=[]):
    ...
No: def foo(a, b=time.time()): # The time the module was loaded???
    ...
No: def foo(a, b=FLAGS.my_thing): # sys.argv has not yet been parsed...
    ...
```

Properties
[link](#)
☐ Use properties for accessing or setting data where you would normally have used simple, lightweight accessor or setter methods.
Definition:

A way to wrap method calls for getting and setting an attribute as a standard attribute access when the computation is lightweight.

Pros:

Readability is increased by eliminating explicit get and set method calls for simple attribute access. Allows calculations to be lazy. Considered the Pythonic way to maintain the interface of a class. In terms of performance, allowing properties bypasses needing trivial accessor methods when a direct variable access is reasonable. This also allows accessor methods to be added in the future without breaking the interface.

Cons:

Properties are specified after the getter and setter methods are declared, requiring one to notice they are used for properties farther down in the code (except for read-only properties created with the `@property` decorator - see below). Must inherit from `object`. Can hide side-effects much like operator overloading. Can be confusing for subclasses.

Decision:

Use properties in new code to access or set data where you would normally have used simple, lightweight accessor or setter methods. Read-only properties should be created with the `@property` [decorator](#).

Inheritance with properties can be non-obvious if the property itself is not overridden. Thus one must make sure that accessor methods are called indirectly to ensure methods overridden in subclasses are called by the property (using the Template Method DP).

```
Yes: import math

class Square(object):
    """A square with two properties: a writable area and a read-only perimeter.

    To use:
    >>> sq = Square(3)
    >>> sq.area
    9
    >>> sq.perimeter
    12
    >>> sq.area = 16
    >>> sq.side
    4
    >>> sq.perimeter
    16
    """
```

```

def __init__(self, side):
    self.side = side

def __get_area(self):
    """Calculates the 'area' property."""
    return self.side ** 2

def __get_area(self):
    """Indirect accessor for 'area' property."""
    return self.__get_area()

def __set_area(self, area):
    """Sets the 'area' property."""
    self.side = math.sqrt(area)

def __set_area(self, area):
    """Indirect setter for 'area' property."""
    self.__set_area(area)

area = property(__get_area, __set_area,
                doc="""Gets or sets the area of the square.""")

@property
def perimeter(self):
    return self.side * 4

```

True/False evaluations

[link](#)

☒ Use the "implicit" false if at all possible.

Definition:

Python evaluates certain values as **false** when in a boolean context. A quick "rule of thumb" is that all "empty" values are considered **false** so **0**, **None**, **[]**, **{}**, **''** all evaluate as **false** in a boolean context.

Pros:

Conditions using Python booleans are easier to read and less error-prone. In most cases, they're also faster.

Cons:

May look strange to C/C++ developers.

Decision:

Use the "implicit" false if at all possible, e.g., **if foo:** rather than **if foo != []:**. There are a few caveats that you should keep in mind though:

- Never use **==** or **!=** to compare singletons like **None**. Use **is** or **is not**.
- 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 be a value that's false in a boolean context!
- Never compare a boolean variable to **False** using **==**. Use **if not x:** instead. If you need to distinguish **False** from **None** then chain the expressions, such as **if not x and x is not None:**.
- For sequences (strings, lists, tuples), use the fact that empty sequences are false, so **if not seq:** or **if seq:** is preferable to **if len(seq):** or **if not len(seq):**.
- When handling integers, implicit false may involve more risk than benefit (i.e., accidentally handling **None** as 0). You may compare a value which is known to be an integer (and is not the result of **len()**) against the integer 0.

```

Yes: if not users:
    print 'no users'

    if foo == 0:
        self.handle_zero()

    if i % 10 == 0:
        self.handle_multiple_of_ten()

```

```

No:  if len(users) == 0:
    print 'no users'

    if foo is not None and not foo:
        self.handle_zero()

    if not i % 10:
        self.handle_multiple_of_ten()

```

- Note that **'0'** (i.e., 0 as string) evaluates to true.

Deprecated Language Features

[link](#)

☒ Use string methods instead of the **string** module where possible. Use function call syntax instead of **apply**. Use list comprehensions and **for** loops instead of **filter** and **map** when the function argument would have been an inlined lambda anyway. Use **for** loops instead of **reduce**.

Definition:

Current versions of Python provide alternative constructs that people find generally preferable.

Decision:

We do not use any Python version which does not support these features, so there is no reason not to use the new styles.

```

Yes: words = foo.split(':')

    [x[1] for x in my_list if x[2] == 5]

    map(math.sqrt, data) # Ok. No inlined lambda expression.

```

```
fn(*args, **kwargs)
```

No: `words = string.split(foo, ':')`

```
map(lambda x: x[1], filter(lambda x: x[2] == 5, my_list))
```

`apply(fn, args, kwargs)`

Lexical Scoping

[link](#)
☐ Okay to use.

Definition:

A nested Python function can refer to variables defined in enclosing functions, but can not assign to them. Variable bindings are resolved using lexical scoping, that is, based on the static program text. Any assignment to a name in a block will cause Python to treat all references to that name as a local variable, even if the use precedes the assignment. If a global declaration occurs, the name is treated as a global variable.

An example of the use of this feature is:

```
def get_adder(summand1):
    """Returns a function that adds numbers to a given number."""
    def adder(summand2):
        return summand1 + summand2

    return adder
```

Pros:

Often results in clearer, more elegant code. Especially comforting to experienced Lisp and Scheme (and Haskell and ML and ...) programmers.

Cons:

Can lead to confusing bugs. Such as this example based on [PEP-0227](#):

```
i = 4
def foo(x):
    def bar():
        print i,
    # ...
    # A bunch of code here
    # ...
    for i in x: # Ah, i *is* local to Foo, so this is what Bar sees
        print i,
    bar()
```

So `foo([1, 2, 3])` will print 1 2 3 3, not 1 2 3 4.

Decision:

Okay to use.

Function and Method Decorators

[link](#)
☐ Use decorators judiciously when there is a clear advantage.

Definition:

[Decorators for Functions and Methods](#) (a.k.a "the @ notation"). The most common decorators are `@classmethod` and `@staticmethod`, for converting ordinary methods to class or static methods. However, the decorator syntax allows for user-defined decorators as well. Specifically, for some function `my_decorator`, this:

```
class C(object):
    @my_decorator
    def method(self):
        # method body ...
```

is equivalent to:

```
class C(object):
    def method(self):
        # method body ...
    method = my_decorator(method)
```

Pros:

Elegantly specifies some transformation on a method; the transformation might eliminate some repetitive code, enforce invariants, etc.

Cons:

Decorators can perform arbitrary operations on a function's arguments or return values, resulting in surprising implicit behavior. Additionally, decorators execute at import time. Failures in decorator code are pretty much impossible to recover from.

Decision:

Use decorators judiciously when there is a clear advantage. Decorators should follow the same import and naming guidelines as functions. Decorator `pydoc` should clearly state that the function is a decorator. Write unit tests for decorators.

Avoid external dependencies in the decorator itself (e.g. don't rely on files, sockets, database connections, etc.), since they might not be available when the decorator runs (at import time, perhaps from `pydoc` or other tools). A decorator that is called with valid parameters should (as much as possible) be guaranteed to succeed in all cases.

Decorators are a special case of "top level code" - see [main](#) for more discussion.

Threading

[link](#) ☐ Do not rely on the atomicity of built-in types.

While Python's built-in data types such as dictionaries appear to have atomic operations, there are corner cases where they aren't atomic (e.g. if `__hash__` or `__eq__` are implemented as Python methods) and their atomicity should not be relied upon. Neither should you rely on atomic variable assignment (since this in turn depends on dictionaries).

Use the Queue module's `Queue` data type as the preferred way to communicate data between threads. Otherwise, use the threading module and its locking primitives. Learn about the proper use of condition variables so you can use `threading.Condition` instead of using lower-level locks.

Power Features

[link](#) ☐ Avoid these features.

Definition:

Python is an extremely flexible language and gives you many fancy features such as metaclasses, access to bytecode, on-the-fly compilation, dynamic inheritance, object reparenting, import hacks, reflection, modification of system internals, etc.

Pros:

These are powerful language features. They can make your code more compact.

Cons:

It's very tempting to use these "cool" features when they're not absolutely necessary. It's harder to read, understand, and debug code that's using unusual features underneath. It doesn't seem that way at first (to the original author), but when revisiting the code, it tends to be more difficult than code that is longer but is straightforward.

Decision:

Avoid these features in your code.

Python Style Rules

Semicolons

[link](#) ☐ Do not terminate your lines with semi-colons and do not use semi-colons to put two commands on the same line.

Line length

[link](#) ☐ Maximum line length is *80 characters*.

Exceptions:

- Long import statements.
- URLs in comments.

Do not use backslash line continuation.

Make use of Python's [implicit line joining inside parentheses, brackets and braces](#). If necessary, you can add an extra pair of parentheses around an expression.

```
Yes: foo_bar(self, width, height, color='black', design=None, x='foo',
           emphasis=None, highlight=0)

      if (width == 0 and height == 0 and
          color == 'red' and emphasis == 'strong'):
```

When a literal string won't fit on a single line, use parentheses for implicit line joining.

```
x = ('This will build a very long long '
     'long long long long long long string')
```

Within comments, put long URLs on their own line if necessary.

```
Yes:  # See details at
      # https://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_name_extension_full_specification.html
```

```
No:   # See details at
      # https://www.example.com/us/developer/documentation/api/content/\
      # v2.0/csv_file_name_extension_full_specification.html
```

Make note of the indentation of the elements in the line continuation examples above; see the [indentation](#) section for explanation.

Parentheses

[link](#) ☐ Use parentheses sparingly.

Do not use them in return statements or conditional statements unless using parentheses for implied line continuation. (See above.) It is however fine to use parentheses around tuples.

```
Yes: if foo:
      bar()
      while x:
          x = bar()
          if x and y:
              bar()
          if not x:
```



```

    bar()
    return foo
    for (x, y) in dict.items(): ...

```

```

No: if (x):
    bar()
    if not(x):
        bar()
    return (foo)

```

Indentation

[link](#)

☒ Indent your code blocks with 4 spaces.

Never use tabs or mix tabs and spaces. In cases of implied line continuation, you should align wrapped elements either vertically, as per the examples in the [line length](#) section; or using a hanging indent of 4 spaces, in which case there should be no argument on the first line.

```

Yes: # Aligned with opening delimiter
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# Aligned with opening delimiter in a dictionary
foo = {
    long_dictionary_key: value1 +
                        value2,
    ...
}

# 4-space hanging indent; nothing on first line
foo = long_function_name(
    var_one, var_two, var_three,
    var_four)

# 4-space hanging indent in a dictionary
foo = {
    long_dictionary_key:
        long_dictionary_value,
    ...
}

```

```

No: # Stuff on first line forbidden
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# 2-space hanging indent forbidden
foo = long_function_name(
    var_one, var_two, var_three,
    var_four)

# No hanging indent in a dictionary
foo = {
    long_dictionary_key:
        long_dictionary_value,
    ...
}

```

Blank Lines

[link](#)

☒ Two blank lines between top-level definitions, one blank line between method definitions.

Two blank lines between top-level definitions, be they function or class definitions. One blank line between method definitions and between the `class` line and the first method. Use single blank lines as you judge appropriate within functions or methods.

Whitespace

[link](#)

☒ Follow standard typographic rules for the use of spaces around punctuation.

No whitespace inside parentheses, brackets or braces.

```

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

```

```

No: spam( ham[ 1 ], { eggs: 2 }, [ ] )

```

No whitespace before a comma, semicolon, or colon. Do use whitespace after a comma, semicolon, or colon except at the end of the line.

```

Yes: if x == 4:
    print x, y
    x, y = y, x

```

```

No: if x == 4 :
    print x , y
    x , y = y , x

```

No whitespace before the open paren/bracket that starts an argument list, indexing or slicing.

```

Yes: spam(1)

```

```

No: spam (1)

```

```

Yes: dict['key'] = list[index]

```

No: `dict ['key'] = list [index]`

Surround binary operators with a single space on either side for assignment (`=`), comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), and Booleans (`and`, `or`, `not`). Use your better judgment for the insertion of spaces around arithmetic operators but always be consistent about whitespace on either side of a binary operator.

Yes: `x == 1`

No: `x<1`

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)`

Don't use spaces to vertically align tokens on consecutive lines, since it becomes a maintenance burden (applies to `:`, `#`, `=`, etc.):

Yes:

```
foo = 1000 # comment
long_name = 2 # comment that should not be aligned

dictionary = {
    'foo': 1,
    'long_name': 2,
```

No:

```
foo      = 1000 # comment
long_name = 2   # comment that should not be aligned

dictionary = {
    'foo'      : 1,
    'long_name': 2,
```

Shebang Line

[link](#)

☒ Most `.py` files do not need to start with a `#!` line. Start the main file of a program with `#!/usr/bin/env python` with an optional single digit 2 or 3 suffix.

This line is used by the kernel to find the Python interpreter, but is ignored by Python when importing modules. It is only necessary on a file that will be executed directly.

Comments

[link](#)

☒ Be sure to use the right style for module, function, method and in-line comments.

Doc Strings

Python has a unique commenting style using doc strings. A doc string is a string that is the first statement in a package, module, class or function. These strings can be extracted automatically through the `__doc__` member of the object and are used by `pydoc`. (Try running `pydoc` on your module to see how it looks.) We always use the three double-quote `"""` format for doc strings (per [PEP 257](#)). A doc string should be organized as a summary line (one physical line) terminated by a period, question mark, or exclamation point, followed by a blank line, followed by the rest of the doc string starting at the same cursor position as the first quote of the first line. There are more formatting guidelines for doc strings below.

Modules

Every file should contain license boilerplate. Choose the appropriate boilerplate for the license used by the project (for example, Apache 2.0, BSD, LGPL, GPL)

Functions and Methods

As used in this section "function" applies to methods, function, and generators.

A function must have a docstring, unless it meets all of the following criteria:

- not externally visible
- very short
- obvious

A docstring should give enough information to write a call to the function without reading the function's code. A docstring should describe the function's calling syntax and its semantics, not its implementation. For tricky code, comments alongside the code are more appropriate than using docstrings.

Certain aspects of a function should be documented in special sections, listed below. Each section begins with a heading line, which ends with a colon. Sections should be indented two spaces, except for the heading.

Args:

List each parameter by name. A description should follow the name, and be separated by a colon and a space. If the description is too long to fit on a single 80-character line, use a hanging indent of 2 or 4 spaces (be consistent with the rest of the file).

The description should mention required type(s) and the meaning of the argument.

If a function accepts `*foo` (variable length argument lists) and/or `**bar` (arbitrary keyword arguments), they should be listed as `*foo` and `**bar`.

Returns: (or Yields: for generators)

Describe the type and semantics of the return value. If the function only returns `None`, this section is not required.

Raises:

List all exceptions that are relevant to the interface.

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
```

```

represented by big_table. Silly things may happen if
other_silly_variable is not None.

Args:
    big_table: An open Bigtable Table instance.
    keys: A sequence of strings representing the key of each table row
          to fetch.
    other_silly_variable: Another optional variable, that has a much
                          longer name than the other args, and which does nothing.

Returns:
    A dict mapping keys to the corresponding table row data
    fetched. Each row is represented as a tuple of strings. For
    example:

    {'Serak': ('Rigel VII', 'Preparer'),
     'Zim': ('Irk', 'Invader'),
     'Lrrr': ('Omicron Persei 8', 'Emperor')}

    If a key from the keys argument is missing from the dictionary,
    then that row was not found in the table.

Raises:
    IOError: An error occurred accessing the bigtable.Table object.
"""
pass

```

Classes

Classes should have a doc string below the class definition describing the class. If your class has public attributes, they should be documented here in an Attributes section and follow the same formatting as a function's Args section.

```

class SampleClass(object):
    """Summary of class here.

    Longer class information...
    Longer class information...

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""

```

Block and Inline Comments

The final place to have comments is in tricky parts of the code. If you're going to have to explain it at the next [code review](#), you should comment it now. Complicated operations get a few lines of comments before the operations commence. Non-obvious ones get comments at the end of the line.

```

# We use a weighted dictionary search to find out where i is in
# the array. We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:           # true iff i is a power of 2

```

To improve legibility, these comments should be at least 2 spaces away from the code.

On the other hand, never describe the code. Assume the person reading the code knows Python (though not what you're trying to do) better than you do.

```

# BAD COMMENT: Now go through the b array and make sure whenever i occurs
# the next element is i+1

```

Classes

[link](#)

▽ If a class inherits from no other base classes, explicitly inherit from `object`. This also applies to nested classes.

```

Yes: class SampleClass(object):
    pass

    class OuterClass(object):
        class InnerClass(object):
            pass

    class ChildClass(ParentClass):
        """Explicitly inherits from another class already."""

```

```

No: class SampleClass:
    pass

```

```
class OuterClass:

    class InnerClass:
        pass
```

Inheriting from `object` is needed to make properties work properly, and it will protect your code from one particular potential incompatibility with Python 3000. It also defines special methods that implement the default semantics of objects including `__new__`, `__init__`, `__delattr__`, `__getattr__`, `__setattr__`, `__hash__`, `__repr__`, and `__str__`.

Strings

[link](#)

- ☒ Use the `format` method or the `%` operator for formatting strings, even when the parameters are all strings. Use your best judgement to decide between `+` and `%` (or `format`) though.

```
Yes: x = a + b
     x = '%s, %s!' % (imperative, expletive)
     x = '{} {}'.format(imperative, expletive)
     x = 'name: %s; score: %d' % (name, n)
     x = 'name: {}; score: {}'.format(name, n)
```

```
No: x = '%s%s' % (a, b) # use + in this case
    x = '{}{}'.format(a, b) # use + in this case
    x = imperative + ', ' + expletive + '!'
    x = 'name: ' + name + '; score: ' + str(n)
```

Avoid using the `+` and `+=` operators to accumulate a string within a loop. Since strings are immutable, this creates unnecessary temporary objects and results in quadratic rather than linear running time. Instead, add each substring to a list and `' '.join` the list after the loop terminates (or, write each substring to a `io.BytesIO` buffer).

```
Yes: items = ['<table>']
     for last_name, first_name in employee_list:
         items.append('<tr><td>%s, %s</td></tr>' % (last_name, first_name))
     items.append('</table>')
     employee_table = ''.join(items)
```

```
No: employee_table = '<table>'
     for last_name, first_name in employee_list:
         employee_table += '<tr><td>%s, %s</td></tr>' % (last_name, first_name)
     employee_table += '</table>'
```

Be consistent with your choice of string quote character within a file. Pick `'` or `"` and stick with it. It is okay to use the other quote character on a string to avoid the need to `\` escape within the string. GPyLint enforces this.

```
Yes:
     Python('Why are you hiding your eyes?')
     Gollum("I'm scared of lint errors.")
     Narrator("Good!" thought a happy Python reviewer.)
```

```
No:
     Python("Why are you hiding your eyes?")
     Gollum('The lint. It burns. It burns us.')
     Gollum("Always the great lint. Watching. Watching.")
```

Prefer `"""` for multi-line strings rather than `'''`. Projects may choose to use `'''` for all non-docstring multi-line strings if and only if they also use `'` for regular strings. Doc strings must use `"""` regardless. Note that it is often cleaner to use implicit line joining since multi-line strings do not flow with the indentation of the rest of the program:

```
Yes:
     print ("This is much nicer.\n"
           "Do it this way.\n")
```

```
No:
     print """This is pretty ugly.
Don't do this.
"""
```

Files and Sockets

[link](#)

- ☒ Explicitly close files and sockets when done with them.

Leaving files, sockets or other file-like objects open unnecessarily has many downsides, including:

- They may consume limited system resources, such as file descriptors. Code that deals with many such objects may exhaust those resources unnecessarily if they're not returned to the system promptly after use.
- Holding files open may prevent other actions being performed on them, such as moves or deletion.
- Files and sockets that are shared throughout a program may inadvertently be read from or written to after logically being closed. If they are actually closed, attempts to read or write from them will throw exceptions, making the problem known sooner.

Furthermore, while files and sockets are automatically closed when the file object is destructed, tying the life-time of the file object to the state of the file is poor practice, for several reasons:

- There are no guarantees as to when the runtime will actually run the file's destructor. Different Python implementations use different memory management techniques, such as delayed Garbage Collection, which may increase the object's lifetime arbitrarily and indefinitely.
- Unexpected references to the file may keep it around longer than intended (e.g. in tracebacks of exceptions, inside globals, etc).

The preferred way to manage files is using the ["with" statement](#):

```
with open("hello.txt") as hello_file:
    for line in hello_file:
```

print line

For file-like objects that do not support the "with" statement, use `contextlib.closing()`:

```
import contextlib

with contextlib.closing(urllib.urlopen("https://www.python.org/")) as front_page:
    for line in front_page:
        print line
```

Legacy AppEngine code using Python 2.5 may enable the "with" statement using "from `__future__` import `with_statement`".

TODO Comments[link](#)

☐ Use **TODO** comments for code that is temporary, a short-term solution, or good-enough but not perfect.

TODOs should include the string **TODO** in all caps, followed by the name, e-mail address, or other identifier of the person who can best provide context about the problem referenced by the **TODO**, in parentheses. A colon is optional. A comment explaining what there is to do is required. The main purpose is to have a consistent **TODO** format that can be searched to find the person who can provide more details upon request. A **TODO** is not a commitment that the person referenced will fix the problem. Thus when you create a **TODO**, it is almost always your name that is given.

```
# TODO(kl@gmail.com): Use a "*" here for string repetition.
# TODO(Zeke) Change this to use relations.
```

If your **TODO** is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2009") or a very specific event ("Remove this code when all clients can handle XML responses.").

Imports formatting[link](#)

☐ Imports should be on separate lines.

E.g.:

```
Yes: import os
     import sys
```

```
No:  import os, sys
```

Imports are always put at the top of the file, just after any module comments and doc strings and before module globals and constants. Imports should be grouped with the order being most generic to least generic:

- standard library imports
- third-party imports
- application-specific imports

Within each grouping, imports should be sorted lexicographically, ignoring case, according to each module's full package path.

```
import foo
from foo import bar
from foo.bar import baz
from foo.bar import Quux
from Foob import ar
```

Statements[link](#)

☐ Generally only one statement per line.

However, you may put the result of a test on the same line as the test only if the entire statement fits on one line. In particular, you can never do so with **try/except** since the **try** and **except** can't both fit on the same line, and you can only do so with an **if** if there is no **else**.

Yes:

```
if foo: bar(foo)
```

No:

```
if foo: bar(foo)
else:  baz(foo)
```

```
try:      bar(foo)
except ValueError: baz(foo)
```

```
try:
    bar(foo)
except ValueError: baz(foo)
```

Access Control[link](#)

☐ If an accessor function would be trivial you should use public variables instead of accessor functions to avoid the extra cost of function calls in Python. When more functionality is added you can use **property** to keep the syntax consistent.

On the other hand, if access is more complex, or the cost of accessing the variable is significant, you should use function calls (following the [Naming](#) guidelines) such as `get_foo()` and `set_foo()`. If the past behavior allowed access through a property, do not bind the new accessor functions to the property. Any code still attempting to access the variable by the old method should break visibly so they are made aware of the change in complexity.

Naming[link](#)

☐ `module_name`, `package_name`, `ClassName`, `method_name`, `ExceptionName`, `function_name`, `GLOBAL_CONSTANT_NAME`, `global_var_name`, `instance_var_name`, `function_parameter_name`, `local_var_name`.

Names to Avoid

- single character names except for counters or iterators
- dashes (-) in any package/module name
- `__double_leading_and_trailing_underscore__` names (reserved by Python)

Naming Convention

- "Internal" means internal to a module or protected or private within a class.
- Prepending a single underscore (`_`) has some support for protecting module variables and functions (not included with `import * from`).
- Prepending a double underscore (`__`) to an instance variable or method effectively serves to make the variable or method private to its class (using name mangling).
- Place related classes and top-level functions together in a module. Unlike Java, there is no need to limit yourself to one class per module.
- Use CapWords for class names, but `lower_with_under.py` for module names. Although there are many existing modules named `CapWords.py`, this is now discouraged because it's confusing when the module happens to be named after a class. ("wait -- did I write `import StringIO` or `from StringIO import StringIO`?")

Guidelines derived from Guido's Recommendations

Type	Public	Internal
Packages	<code>lower_with_under</code>	
Modules	<code>lower_with_under</code>	<code>_lower_with_under</code>
Classes	<code>CapWords</code>	<code>_CapWords</code>
Exceptions	<code>CapWords</code>	
Functions	<code>lower_with_under()</code>	<code>_lower_with_under()</code>
Global/Class Constants	<code>CAPS_WITH_UNDER</code>	<code>_CAPS_WITH_UNDER</code>
Global/Class Variables	<code>lower_with_under</code>	<code>_lower_with_under</code>
Instance Variables	<code>lower_with_under</code>	<code>_lower_with_under</code> (protected) or <code>__lower_with_under</code> (private)
Method Names	<code>lower_with_under()</code>	<code>_lower_with_under()</code> (protected) or <code>__lower_with_under()</code> (private)
Function/Method Parameters	<code>lower_with_under</code>	
Local Variables	<code>lower_with_under</code>	

Main

[link](#)

- ☒ Even a file meant to be used as a script should be importable and a mere import should not have the side effect of executing the script's main functionality. The main functionality should be in a `main()` function.

In Python, `pydoc` as well as unit tests require modules to be importable. Your code should always check `if __name__ == '__main__':` before executing your main program so that the main program is not executed when the module is imported.

```
def main():
    ...

if __name__ == '__main__':
    main()
```

All code at the top level will be executed when the module is imported. Be careful not to call functions, create objects, or perform other operations that should not be executed when the file is being `pydoc`ed.

Parting Words

BE CONSISTENT.

If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around all their arithmetic operators, you should too. If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you're saying rather than on how you're saying it. We present global style rules here so people know the vocabulary, but local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Avoid this.

Revision 2.59

Amit Patel
Antoine Picard
Eugene Jhong
Gregory P. Smith
Jeremy Hylton
Matt Smart
Mike Shields
Shane Liebling