Google Python Style Guide

September 4, 2024

Abstract

The Google Python Style Guide is a comprehensive set of best practices and conventions designed to ensure consistency, readability, and maintainability in Python code. It covers a wide range of topics, including naming conventions, code formatting, documentation, and programming practices. The guide emphasizes the importance of writing clear, concise, and efficient code, and provides practical advice on how to achieve this. By adhering to the guidelines outlined in the Google Python Style Guide, developers can create code that is not only functional but also easy to understand and maintain, thereby promoting collaboration and reducing the likelihood of errors. This document serves as a valuable resource for both individual developers and teams looking to adopt a standardized approach to Python programming.

Contents

1 Background

Python is the main dynamic 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.

Many teams use the Black or Pyink auto-formatter to avoid arguing over formatting.

2 Python Language Rules

2.1 Lint

Run `pylint` over your code using this pylintrc.

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, sometimes we'll need to write around it, suppress its warnings or fix 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:

```
def do_PUT(self):  # WSGI name, so pylint: disable=invalid-name
  ...
```

`pylint` warnings are each identified by symbolic name (`empty-docstring`) Google-specific warnings start with `g-`.

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=invalid-name
```

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

Unused argument warnings can be suppressed by deleting the variables at the beginning of the function. Always include a comment explaining why you are deleting it. "Unused." is sufficient. For example:

```
def viking_cafe_order(spam: str, beans: str, eggs: str | None = None) -> str:
    del beans, eggs  # Unused by vikings.
    return spam + spam + spam
```

Other common forms of suppressing this warning include using '`_`' as the identifier for the unused argument or prefixing the argument name with '`unused_`', or assigning them to '`_`'. These forms are allowed but no longer encouraged. These break callers that pass arguments by name and do not enforce that the arguments are actually unused.

2.2 Imports

Use `import` statements for packages and modules only, not for individual types, classes, or functions.

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` in any of the following circumstances:
  - Two modules named `y` are to be imported.
  - `y` conflicts with a top-level name defined in the current module.
  - `y` conflicts with a common parameter name that is part of the public API (e.g., `features`).
  - `y` is an inconveniently long name.
  - `y` is too generic in the context of your code (e.g., `from storage.file_system import options as fs_options`).
- Use `import y as z` only when `z` is a standard abbreviation (e.g., `import numpy as np`).

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.

Exemptions Exemptions from this rule:

- Symbols from the following modules are used to support static analysis and type checking:
  - `typing` module
  - `collections.abc` module
  - `typing_extensions` module
- Redirects from the six.moves module.

2.3 Packages

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

Pros Avoids conflicts in module names or incorrect imports due to the module search path not being what the author expected. Makes it easier to find modules.

Cons Makes it harder to deploy code because you have to replicate the package hierarchy. Not really a problem with modern deployment mechanisms.

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

Imports should be as follows:

```
Yes:
  # Reference absl.flags in code with the complete name (verbose).
  import absl.flags
  from doctor.who import jodie

  _FOO = absl.flags.DEFINE_string(...)
```

```
Yes:
  # Reference flags in code with just the module name (common).
  from absl import flags
  from doctor.who import jodie

  _FOO = flags.DEFINE_string(...)
```

(assume this file lives in `doctor/who/` where `jodie.py` also exists)

```
No:
  # Unclear what module the author wanted and what will be imported.  The actual
  # import behavior depends on external factors controlling sys.path.
  # Which possible jodie module did the author intend to import?
  import jodie
```

The directory the main binary is located in should not be assumed to be in `sys.path` despite that happening in some environments. This being the case, code should assume that `import jodie` refers to a third-party or top-level package named `jodie`, not a local `jodie.py`.

2.4 Exceptions

Exceptions are allowed but must be used carefully.

Definition Exceptions are a means of breaking out of normal control flow 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 plumb error codes through.

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

Decision Exceptions must follow certain conditions:

- Make use of built-in exception classes when it makes sense. For example, raise a `ValueError` to indicate a programming mistake like a violated precondition, such as may happen when validating function arguments.
- Do not use `assert` statements in place of conditionals or validating preconditions. They must not be critical to the application logic. A litmus test would be that the `assert` could be removed without breaking the code. `assert` conditionals are not guaranteed to be evaluated. For pytest based tests,

`assert` is okay and expected to verify expectations. For example:

Yes:

```
  def connect_to_next_port(self, minimum: int) -> int:
    """Connects to the next available port.

    Args:
      minimum: A port value greater or equal to 1024.

    Returns:
      The new minimum port.

    Raises:
      ConnectionError: If no available port is found.
    """
    if minimum < 1024:
      # Note that this raising of ValueError is not mentioned in the doc
      # string's "Raises:" section because it is not appropriate to
      # guarantee this specific behavioral reaction to API misuse.
      raise ValueError(f'Min. port must be at least 1024, not {minimum}.')
    port = self._find_next_open_port(minimum)
    if port is None:
      raise ConnectionError(
          f'Could not connect to service on port {minimum} or higher.')
    # The code does not depend on the result of this assert.
    assert port >= minimum, (
        f'Unexpected port {port} when minimum was {minimum}.')
    return port
```

No:

```
  def connect_to_next_port(self, minimum: int) -> int:
    """Connects to the next available port.

    Args:
      minimum: A port value greater or equal to 1024.

    Returns:
      The new minimum port.
    """
    assert minimum >= 1024, 'Minimum port must be at least 1024.'
    # The following code depends on the previous assert.
    port = self._find_next_open_port(minimum)
    assert port is not None
    # The type checking of the return statement relies on the assert.
    return port
```

- Libraries or packages may define their own exceptions. When doing so they must inherit from an existing exception class. Exception names should end in `Error` and should not introduce repetition (`foo.FooError`).
- Never use catch-all `except:` statements, or catch `Exception` or `StandardError`, unless you are
  - re-raising the exception, or
  - creating an isolation point in the program where exceptions are not propagated but are recorded and suppressed instead, such as protecting a thread from crashing by guarding its outermost block.

  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.

2.5 Mutable

Global State

Avoid mutable global state.

Definition Module-level values or class attributes that can get mutated during program execution.

Pros Occasionally useful.

Cons

- Breaks encapsulation: Such design can make it hard to achieve valid objectives. For example, if global state is used to manage a database connection, then connecting to two different databases at the same time (such as for computing differences during a migration) becomes difficult. Similar problems easily arise with global registries.
- Has the potential to change module behavior during the import, because assignments to global variables are done when the module is first imported.

Decision Avoid mutable global state.

In those rare cases where using global state is warranted, mutable global entities should be declared at the module level or as a class attribute and made internal by prepending an `_` to the name. If necessary, external access to mutable global state must be done through public functions or class methods. See Naming below. Please explain the design reasons why mutable global state is being used in a comment or a doc linked to from a comment.

Module-level constants are permitted and encouraged. For example: `_MAX_HOLY_HANDGRENADE_COUNT = 3` for an internal use constant or `SIR_LANCELOTS_FAVORITE_COLOR = "blue"` for a public API constant. Constants must be named using all caps with underscores. See Naming below.

2.6 Nested/Local/Inner Classes and Functions

Nested local functions or classes are fine when used to close over a local variable. Inner classes 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 ADT-y. Commonly used for implementing decorators.

Cons Nested functions and classes cannot be directly tested. Nesting can make the outer function longer and less readable.

Decision They are fine with some caveats. Avoid nested functions or classes except when closing over a local value other than `self` or `cls`. Do not nest a function just to hide it from users of a module. Instead, prefix its name with an _ at the module level so that it can still be accessed by tests.

2.7 Comprehensions & Generator Expressions

Okay to use for simple cases.

Definition List, Dict, and Set comprehensions as well as generator expressions provide a concise and efficient way to create container types and iterators without resorting to the use of traditional loops, `map()`, `filter()`, or `lambda`.

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

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

Decision Comprehensions are allowed, however multiple `for` clauses or filter expressions are not permitted. Optimize for readability, not conciseness.

Yes:

```
  result = [mapping_expr for value in iterable if filter_expr]

  result = [
      is_valid(metric={'key': value})
      for value in interesting_iterable
      if a_longer_filter_expression(value)
  ]

  descriptive_name = [
      transform({'key': key, 'value': value}, color='black')
      for key, value in generate_iterable(some_input)
      if complicated_condition_is_met(key, value)
  ]

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

  return {
      x: complicated_transform(x)
      for x in long_generator_function(parameter)
      if x is not None
  }

  return (x**2 for x in range(10))

  unique_names = {user.name for user in users if user is not None}
```

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 range(5)
      for y in range(5)
      if x != y
      for z in range(5)
      if y != z
  )
```

2.8 Default

Iterators and Operators

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 (unless the variable has type annotations). 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 obj in alist: ...
      for line in afile: ...
      for k, v in adict.items(): ...

No:   for key in adict.keys(): ...
      for line in afile.readlines(): ...
```

2.9 Generators

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 Local variables in the generator will not be garbage collected until the generator is either consumed to exhaustion or itself garbage collected.

Decision Fine. Use "Yields:" rather than "Returns:" in the docstring for generator functions.

If the generator manages an expensive resource, make sure to force the clean up.

A good way to do the clean up is by wrapping the generator with a context manager PEP-0533.

2.10 Lambda Functions

Okay for one-liners. Prefer generator expressions over `map()` or `filter()` with a `lambda`.

Definition Lambdas define anonymous functions in an expression, as opposed to a statement.

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 Lambdas are allowed. If the code inside the lambda function spans multiple lines or is longer than 60-80 chars, it might be 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`.

2.11 Conditional Expressions

Okay for simple cases.

Definition Conditional expressions (sometimes called a "ternary operator") 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 simple cases. Each portion must fit on one line: true-expression, if-expression, else-expression. Use a complete if statement when things get more complicated.

```
Yes:
    one_line = 'yes' if predicate(value) else 'no'
    slightly_split = ('yes' if predicate(value)
                      else 'no, nein, nyet')
    the_longest_ternary_style_that_can_be_done = (
        'yes, true, affirmative, confirmed, correct'
        if predicate(value)
        else 'no, false, negative, nay')
```

```
No:
    bad_line_breaking = ('yes' if predicate(value) else
                         'no')
    portion_too_long = ('yes'
                        if some_long_module.some_long_predicate_function(
                            really_long_variable_name)
                        else 'no, false, negative, nay')
```

2.12 Default Argument Values

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 on rare occasions 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. As Python does not support overloaded methods/functions, 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 = []
Yes: def foo(a, b: Sequence | None = None):
         if b is None:
             b = []
Yes: def foo(a, b: Sequence = ()):  # Empty tuple OK since tuples are immutable.
         ...

from absl import flags
_FOO = flags.DEFINE_string(...)

No:  def foo(a, b=[]):
         ...
No:  def foo(a, b=time.time()):  # Is `b` supposed to represent when this module was loaded?
         ...
No:  def foo(a, b=_FOO.value):  # sys.argv has not yet been parsed...
         ...
No:  def foo(a, b: Mapping = {}):  # Could still get passed to unchecked code.
         ...
```

2.13 Properties

Properties may be used to control getting or setting attributes that require trivial computations or logic. Property implementations must match the general expectations of regular attribute access: that they are cheap, straightforward, and unsurprising.

Definition A way to wrap method calls for getting and setting an attribute as a standard attribute access.

Pros

- Allows for an attribute access and assignment API rather than getter and setter method calls.
- Can be used to make an attribute read-only.
- Allows calculations to be lazy.
- Provides a way to maintain the public interface of a class when the internals evolve independently of class users.

Cons

- Can hide side-effects much like operator overloading.
- Can be confusing for subclasses.

Decision Properties are allowed, but, like operator overloading, should only be used when necessary and match the expectations of typical attribute access; follow the getters and setters rules otherwise.

For example, using a property to simply both get and set an internal attribute isn't allowed: there is no computation occurring, so the property is unnecessary (make the attribute public instead). In comparison, using a property to control attribute access or to calculate a *trivially* derived value is allowed: the logic is simple and unsurprising.

Properties should be created with the `@property` decorator. Manually implementing a property descriptor is considered a power feature.

Inheritance with properties can be non-obvious. Do not use properties to implement computations a subclass may ever want to override and extend.

2.14 True/False Evaluations

Use the "implicit" false if at all possible (with a few caveats).

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 possible, e.g., `if foo:` rather than `if foo != []:`. There are a few caveats that you should keep in mind though:

- Always use `if foo is None:` (or `is not None`) to check for a `None` value. 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 seq:` and `if not seq:` are preferable to `if len(seq):` and `if not len(seq):` respectively.
- 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 i % 10 == 0:
      self.handle_multiple_of_ten()

  def f(x=None):
      if x is None:
          x = []
  ```

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

  if not i % 10:
      self.handle_multiple_of_ten()

  def f(x=None):
      x = x or []
  ```

- Note that `'0'` (i.e., `0` as string) evaluates to true.
- Note that Numpy arrays may raise an exception in an implicit boolean context. Prefer the `.size` attribute when testing emptiness of a `np.array` (e.g. `if not users.size`).

2.15 Lexical Scoping

Okay to use.

Definition A nested Python function can refer to variables defined in enclosing functions, but cannot 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: float) -> Callable[[float], float]:
    """Returns a function that adds numbers to a given number."""
    def adder(summand2: float) -> float:
        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: Iterable[int]):
    def bar():
        print(i, end='')
    # ...
    # A bunch of code here
    # ...
    for i in x:  # Ah, i *is* local to foo, so this is what bar sees
        print(i, end='')
    bar()
```

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

Decision Okay to use.

2.16 Function and Method Decorators

Use decorators judiciously when there is a clear advantage. Avoid `staticmethod` and limit use of `classmethod`.

Definition Decorators for Functions and Methods (a.k.a "the `@` notation"). One common decorator is `@property`, used for converting ordinary methods into dynamically computed attributes. However, the decorator syntax allows for user-defined decorators as well. Specifically, for some function `my_decorator`, this:

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

is equivalent to:

```
class C:
    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 object definition time. For module-level objects (classes, module functions, ...) this happens 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.

Never use `staticmethod` unless forced to in order to integrate with an API defined in an existing library. Write a module-level function instead.

Use `classmethod` only when writing a named constructor, or a class-specific routine that modifies necessary global state such as a process-wide cache.

2.17 Threading

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. Prefer condition variables and `threading.Condition` instead of using lower-level locks.

2.18 Power

Features

Avoid these features.

Definition Python is an extremely flexible language and gives you many fancy features such as custom metaclasses, access to bytecode, on-the-fly compilation, dynamic inheritance, object reparenting, import hacks, reflection (e.g. some uses of `getattr()`), modification of system internals, `__del__` methods implementing customized cleanup, 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.

Standard library modules and classes that internally use these features are okay to use (for example, `abc.ABCMeta`, `dataclasses`, and `enum`).

2.19 Modern

Python: from ___future___ imports

New language version semantic changes may be gated behind a special future import to enable them on a per-file basis within earlier runtimes.

Definition Being able to turn on some of the more modern features via `from __future__ import` statements allows early use of features from expected future Python versions.

Pros This has proven to make runtime version upgrades smoother as changes can be made on a per-file basis while declaring compatibility and preventing regressions within those files. Modern code is more maintainable as it is less likely to accumulate technical debt that will be problematic during future runtime upgrades.

Cons Such code may not work on very old interpreter versions prior to the introduction of the needed future statement. The need for this is more common in projects supporting an extremely wide variety of environments.

Decision

from ___future___ imports Use of `from __future__ import` statements is encouraged. It allows a given source file to start using more modern Python syntax features today. Once you no longer need to run on a version where the features are hidden behind a `__future__` import, feel free to remove those lines.

In code that may execute on versions as old as 3.5 rather than >= 3.7, import:

```
from __future__ import generator_stop
```

For more information read the Python future statement definitions documentation.

Please don't remove these imports until you are confident the code is only ever used in a sufficiently modern environment. Even if you do not currently use the feature a specific future import enables in your code today, keeping it in place in the file prevents later modifications of the code from inadvertently depending on the older behavior.

Use other `from __future__` import statements as you see fit.

2.20 Type Annotated Code

You can annotate Python code with type hints according to PEP-484, and type-check the code at build time with a type checking tool like pytype.

Type annotations can be in the source or in a stub pyi file. Whenever possible, annotations should be in the source. Use pyi files for third-party or extension modules.

Definition Type annotations (or "type hints") are for function or method arguments and return values:

```
def func(a: int) -> list[int]:
```

You can also declare the type of a variable using similar PEP-526 syntax:

```
a: SomeType = some_func()
```

Pros Type annotations improve the readability and maintainability of your code. The type checker will convert many runtime errors to build-time errors, and reduce your ability to use Power Features.

Cons You will have to keep the type declarations up to date. You might see type errors that you think are valid code. Use of a type checker may reduce your ability to use Power Features.

Decision You are strongly encouraged to enable Python type analysis when updating code. When adding or modifying public APIs, include type annotations and enable checking via pytype in the build system. As static analysis is relatively new to Python, we acknowledge that undesired side-effects (such as wrongly inferred types) may prevent adoption by some projects. In those situations, authors are encouraged to add a comment with a TODO or link to a bug describing the issue(s) currently preventing type annotation adoption in the BUILD file or in the code itself as appropriate.

3 Python Style Rules

3.1 Semicolons

Do not terminate your lines with semicolons, and do not use semicolons to put two statements on the same line.

3.2 Line

length

Maximum line length is *80 characters*.

Explicit exceptions to the 80 character limit:

- Long import statements.
- URLs, pathnames, or long flags in comments.
- Long string module-level constants not containing whitespace that would be inconvenient to split across lines such as URLs or pathnames.
  - Pylint disable comments. (e.g.: `# pylint: disable=invalid-name`)

Do not use a backslash for explicit line continuation.

Instead, 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.

Note that this rule doesn't prohibit backslash-escaped newlines within strings (see below).

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


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

     (bridge_questions.clarification_on
      .average_airspeed_of.unladen_swallow) = 'African or European?'

     with (
         very_long_first_expression_function() as spam,
         very_long_second_expression_function() as beans,
         third_thing() as eggs,
     ):
       place_order(eggs, beans, spam, beans)


No:  if width == 0 and height == 0 and \
         color == 'red' and emphasis == 'strong':

     bridge_questions.clarification_on \
         .average_airspeed_of.unladen_swallow = 'African or European?'

     with very_long_first_expression_function() as spam, \
           very_long_second_expression_function() as beans, \
           third_thing() as eggs:
       place_order(eggs, beans, spam, beans)
```

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')
```

Prefer to break lines at the highest possible syntactic level. If you must break a line twice, break it at the

same syntactic level both times.

```
Yes: bridgekeeper.answer(
         name="Arthur", quest=questlib.find(owner="Arthur", perilous=True))

     answer = (a_long_line().of_chained_methods()
               .that_eventually_provides().an_answer())

     if (
         config is None
         or 'editor.language' not in config
         or config['editor.language'].use_spaces is False
     ):
       use_tabs()
```

```
No: bridgekeeper.answer(name="Arthur", quest=questlib.find(
        owner="Arthur", perilous=True))

    answer = a_long_line().of_chained_methods().that_eventually_provides(
        ).an_answer()

    if (config is None or 'editor.language' not in config or config[
        'editor.language'].use_spaces is False):
      use_tabs()
```

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

```
Yes:  # See details at
      # http://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_name_extension_full_
```

```
No:  # See details at
     # http://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.

In all other cases where a line exceeds 80 characters, and the Black or Pyink auto-formatter does not help bring the line below the limit, the line is allowed to exceed this maximum. Authors are encouraged to manually break the line up per the notes above when it is sensible.

3.3 Parentheses

Use parentheses sparingly.

It is fine, though not required, to use parentheses around tuples. Do not use them in return statements or

conditional statements unless using parentheses for implied line continuation or to indicate a tuple.

```
Yes: if foo:
         bar()
     while x:
         x = bar()
     if x and y:
         bar()
     if not x:
         bar()
     # For a 1 item tuple the ()s are more visually obvious than the comma.
     onesie = (foo,)
     return foo
     return spam, beans
     return (spam, beans)
     for (x, y) in dict.items(): ...
```

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

3.4 Indentation

Indent your code blocks with *4 spaces*.

Never use tabs. Implied line continuation should align wrapped elements vertically (see line length examples), or use a hanging 4-space indent. Closing (round, square or curly) brackets can be placed at the end of the expression, or on separate lines, but then should be indented the same as the line with the corresponding

opening bracket.

```
Yes:    # Aligned with opening delimiter.
        foo = long_function_name(var_one, var_two,
                                 var_three, var_four)
        meal = (spam,
                beans)

        # 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)
        meal = (
            spam,
            beans)

        # 4-space hanging indent; nothing on first line,
        # closing parenthesis on a new line.
        foo = long_function_name(
            var_one, var_two, var_three,
            var_four
        )
        meal = (
            spam,
            beans,
        )

        # 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)
        meal = (spam,
            beans)

        # 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,
            ...
        }
```

Trailing commas in sequences of items? Trailing commas in sequences of items are recommended only when the closing container token `]`, `)`, or `}` does not appear on the same line as the final element, as well as for tuples with a single element. The presence of a trailing comma is also used as a hint to our Python code auto-formatter Black or Pyink to direct it to auto-format the container of items to one item per line when the `,` after the final element is present.

```
Yes:   golomb3 = [0, 1, 3]
       golomb4 = [
           0,
           1,
           4,
           6,
       ]

No:    golomb4 = [
           0,
           1,
           4,
           6,]
```

3.5 Blank Lines

Two blank lines between top-level definitions, be they function or class definitions. One blank line between method definitions and between the docstring of a `class` and the first method. No blank line following a `def` line. Use single blank lines as you judge appropriate within functions or methods.

Blank lines need not be anchored to the definition. For example, related comments immediately preceding function, class, and method definitions can make sense. Consider if your comment might be more useful as part of the docstring.

3.6 Whitespace

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]
```

No trailing whitespace.

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 (+, -, *, /, //, %, **, @).

```
Yes: x == 1
```

```
No:  x<1
```

Never use spaces around = when passing keyword arguments or defining a default parameter value, with one exception: when a type annotation is present, *do* use spaces around the = for the default parameter value.

```
Yes: def complex(real, imag=0.0): return Magic(r=real, i=imag)
Yes: def complex(real, imag: float = 0.0): return Magic(r=real, i=imag)
```

```
No:  def complex(real, imag = 0.0): return Magic(r = real, i = imag)
No:  def complex(real, imag: float=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,
  }
```

3.7 Shebang Line

Most .py files do not need to start with a #! line. Start the main file of a program with #!/usr/bin/env python3 (to support virtualenvs) or #!/usr/bin/python3 per PEP-394.

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 intended to be executed directly.

3.8 Comments and Docstrings

Be sure to use the right style for module, function, method docstrings and inline comments.

Docstrings Python uses *docstrings* to document code. A docstring 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.) Always use the three-double-quote """ format for docstrings (per PEP 257). A docstring should be organized as a summary line (one physical line not exceeding 80 characters) terminated by a period, question mark, or exclamation point. When writing more (encouraged), this must be followed by a blank line, followed by the rest of the docstring starting at the same cursor position as the first quote of the first line. There are more formatting guidelines for docstrings 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).

Files should start with a docstring describing the contents and usage of the module.

```
"""A one-line summary of the module or program, terminated by a period.

Leave one blank line.  The rest of this docstring should contain an
overall description of the module or program.  Optionally, it may also
contain a brief description of exported classes and functions and/or usage
examples.

Typical usage example:

  foo = ClassFoo()
  bar = foo.FunctionBar()
"""
```

Test modules Module-level docstrings for test files are not required. They should be included only when there is additional information that can be provided.

Examples include some specifics on how the test should be run, an explanation of an unusual setup pattern, dependency on the external environment, and so on.

```
"""This blaze test uses golden files.

You can update those files by running
`blaze run //foo/bar:foo_test -- --update_golden_files` from the `google3`
directory.
"""
```

Docstrings that do not provide any new information should not be used.

```
"""Tests for foo.bar."""
```

Functions and Methods In this section, "function" means a method, function, generator, or property.

A docstring is mandatory for every function that has one or more of the following properties:

- being part of the public API
- nontrivial size
- non-obvious logic

A docstring should give enough information to write a call to the function without reading the function's code. The docstring should describe the function's calling syntax and its semantics, but generally not its implementation details, unless those details are relevant to how the function is to be used. For example, a function that mutates one of its arguments as a side effect should note that in its docstring. Otherwise, subtle but important details of a function's implementation that are not relevant to the caller are better expressed as comments alongside the code than within the function's docstring.

The docstring may be descriptive-style (`"""Fetches rows from a Bigtable."""`) or imperative-style (`"""Fetch rows from a Bigtable."""`), but the style should be consistent within a file. The docstring for a `@property` data descriptor should use the same style as the docstring for an attribute or a function argument (`"""The Bigtable path."""`, rather than `"""Returns the Bigtable path."""`).

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. All sections other than the heading should maintain a hanging indent of two or four spaces (be consistent within a file). These sections can be omitted in cases where the function's name and signature are informative enough that it can be aptly described using a one-line docstring.

Args: : List each parameter by name. A description should follow the name, and be separated by a colon followed by either a space or newline. If the description is too long to fit on a single 80-character line, use a hanging indent of 2 or 4 spaces more than the parameter name (be consistent with the rest of the docstrings in the file). The description should include required type(s) if the code does not contain a corresponding type annotation. 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 semantics of the return value, including any type information that the type annotation does not provide. If the function only returns None, this section is not required. It may also be omitted if the docstring starts with "Return", "Returns", "Yield", or "Yields" (e.g. `"""Returns row from Bigtable as a tuple of strings."""`) *and* the opening sentence is sufficient to describe the return value. Do not imitate older 'NumPy style' (example), which frequently documented a tuple return value as if it were multiple return values with individual names (never mentioning the tuple). Instead, describe such a return value as: "Returns: A tuple (mat_a, mat_b), where mat_a is ..., and ...". The auxiliary names in the docstring need not necessarily correspond to any internal names used in the function body (as those are not part of the API). If the function uses `yield` (is a generator), the `Yields:` section should document the object returned by `next()`, instead of the generator object itself that the call evaluates to.

Raises: : List all exceptions that are relevant to the interface followed by a description. Use a similar exception name + colon + space or newline and hanging indent style as described in *Args:*. You should not document exceptions that get raised if the API specified in the docstring is violated (because this would paradoxically make behavior under violation of the API part of the API).

```
def fetch_smalltable_rows(
    table_handle: smalltable.Table,
    keys: Sequence[bytes | str],
    require_all_keys: bool = False,
) -> Mapping[bytes, tuple[str, ...]]:
    """Fetches rows from a Smalltable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by table_handle.  String keys will be UTF-8 encoded.

    Args:
        table_handle: An open smalltable.Table instance.
        keys: A sequence of strings representing the key of each table
          row to fetch.  String keys will be UTF-8 encoded.
        require_all_keys: If True only rows with values set for all keys will be
          returned.

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

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

        Returned keys are always bytes.  If a key from the keys argument is
        missing from the dictionary, then that row was not found in the
        table (and require_all_keys must have been False).

    Raises:
        IOError: An error occurred accessing the smalltable.
    """
```

Similarly, this variation on `Args:` with a line break is also allowed:

```
def fetch_smalltable_rows(
    table_handle: smalltable.Table,
    keys: Sequence[bytes | str],
    require_all_keys: bool = False,
) -> Mapping[bytes, tuple[str, ...]]:
    """Fetches rows from a Smalltable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by table_handle.  String keys will be UTF-8 encoded.

    Args:
      table_handle:
        An open smalltable.Table instance.
      keys:
        A sequence of strings representing the key of each table row to
        fetch.  String keys will be UTF-8 encoded.
      require_all_keys:
        If True only rows with values set for all keys will be returned.

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

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

      Returned keys are always bytes.  If a key from the keys argument is
      missing from the dictionary, then that row was not found in the
      table (and require_all_keys must have been False).

    Raises:
      IOError: An error occurred accessing the smalltable.
    """
```

Overridden Methods A method that overrides a method from a base class does not need a docstring if it is explicitly decorated with `@override` (from `typing_extensions` or `typing` modules), unless the overriding method's behavior materially refines the base method's contract, or details need to be provided (e.g., documenting additional side effects), in which case a docstring with at least those differences is required

on the overriding method.

```
from typing_extensions import override

class Parent:
  def do_something(self):
    """Parent method, includes docstring."""

# Child class, method annotated with override.
class Child(Parent):
  @override
  def do_something(self):
    pass

# Child class, but without @override decorator, a docstring is required.
class Child(Parent):
  def do_something(self):
    pass

# Docstring is trivial, @override is sufficient to indicate that docs can be
# found in the base class.
class Child(Parent):
  @override
  def do_something(self):
    """See base class."""
```

Classes Classes should have a docstring below the class definition describing the class. Public attributes, excluding properties, should be documented here in an `Attributes` section and follow the same formatting as a function's `Args` section.

```
class SampleClass:
    """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: bool = False):
        """Initializes the instance based on spam preference.

        Args:
          likes_spam: Defines if instance exhibits this preference.
        """
        self.likes_spam = likes_spam
        self.eggs = 0

    @property
    def butter_sticks(self) -> int:
        """The number of butter sticks we have."""
```

All class docstrings should start with a one-line summary that describes what the class instance represents. This implies that subclasses of `Exception` should also describe what the exception represents, and not the

context in which it might occur. The class docstring should not repeat unnecessary information, such as that the class is a class.

```
# Yes:
class CheeseShopAddress:
  """The address of a cheese shop.

  ...
  """

class OutOfCheeseError(Exception):
  """No more cheese is available."""
```

```
# No:
class CheeseShopAddress:
  """Class that describes the address of a cheese shop.

  ...
  """

class OutOfCheeseError(Exception):
  """Raised when no more cheese is available."""
```

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 if i is 0 or a power of 2.
```

To improve legibility, these comments should start at least 2 spaces away from the code with the comment character `#`, followed by at least one space before the text of the comment itself.

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
```

Punctuation, Spelling, and Grammar Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalization and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

3.9 Strings

Use an f-string, the `%` operator, or the `format` method for formatting strings, even when the parameters are all strings. Use your best judgment to decide between string formatting options. A single join with `+` is okay

but do not format with `+`.

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

No: x = first + ', ' + second
    x = 'name: ' + name + '; score: ' + str(n)
```

Avoid using the `+` and `+=` operators to accumulate a string within a loop. In some conditions, accumulating a string with addition can lead to quadratic rather than linear running time. Although common accumulations of this sort may be optimized on CPython, that is an implementation detail. The conditions under which an optimization applies are not easy to predict and may change. Instead, add each substring to a list and `''.join` the list after the loop terminates, or write each substring to an `io.StringIO` buffer. These techniques consistently have amortized-linear run-time complexity.

```
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 backslash-escape quote characters within the string.

```
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. Docstrings must use `"""` regardless.

Multi-line strings do not flow with the indentation of the rest of the program. If you need to avoid embedding extra space in the string, use either concatenated single-line strings or a multi-line string with

`textwrap.dedent()` to remove the initial space on each line:

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

  Yes:
  long_string = """This is fine if your use case can accept
      extraneous leading spaces."""

  Yes:
  long_string = ("And this is fine if you cannot accept\n" +
                 "extraneous leading spaces.")

  Yes:
  long_string = ("And this too is fine if you cannot accept\n"
                 "extraneous leading spaces.")

  Yes:
  import textwrap

  long_string = textwrap.dedent("""\
      This is also fine, because textwrap.dedent()
      will collapse common leading spaces in each line.""")
```

Note that using a backslash here does not violate the prohibition against explicit line continuation; in this case, the backslash is escaping a newline in a string literal.

Logging For logging functions that expect a pattern-string (with %-placeholders) as their first argument: Always call them with a string literal (not an f-string!) as their first argument with pattern-parameters as subsequent arguments. Some logging implementations collect the unexpanded pattern-string as a queryable

field. It also prevents spending time rendering a message that no logger is configured to output.

Yes:

```
import tensorflow as tf
logger = tf.get_logger()
logger.info('TensorFlow Version is: %s', tf.__version__)
```

Yes:

```
import os
from absl import logging

logging.info('Current $PAGER is: %s', os.getenv('PAGER', default=''))

homedir = os.getenv('HOME')
if homedir is None or not os.access(homedir, os.W_OK):
  logging.error('Cannot write to home directory, $HOME=%r', homedir)
```

No:

```
import os
from absl import logging

logging.info('Current $PAGER is:')
logging.info(os.getenv('PAGER', default=''))

homedir = os.getenv('HOME')
if homedir is None or not os.access(homedir, os.W_OK):
  logging.error(f'Cannot write to home directory, $HOME={homedir!r}')
```

Error Messages Error messages (such as: message strings on exceptions like `ValueError`, or messages shown to the user) should follow three guidelines:

1. The message needs to precisely match the actual error condition.
2. Interpolated pieces need to always be clearly identifiable as such.
3. They should allow simple automated processing (e.g. grepping).

```
Yes:
if not 0 <= p <= 1:
  raise ValueError(f'Not a probability: {p=}')

try:
  os.rmdir(workdir)
except OSError as error:
  logging.warning('Could not remove directory (reason: %r): %r',
                  error, workdir)
```

```
No:
if p < 0 or p > 1:  # PROBLEM: also false for float('nan')!
  raise ValueError(f'Not a probability: {p=}')

try:
  os.rmdir(workdir)
except OSError:
  # PROBLEM: Message makes an assumption that might not be true:
  # Deletion might have failed for some other reason, misleading
  # whoever has to debug this.
  logging.warning('Directory already was deleted: %s', workdir)

try:
  os.rmdir(workdir)
except OSError:
  # PROBLEM: The message is harder to grep for than necessary, and
  # not universally non-confusing for all possible values of `workdir`.
  # Imagine someone calling a library function with such code
  # using a name such as workdir = 'deleted'. The warning would read:
  # "The deleted directory could not be deleted."
  logging.warning('The %s directory could not be deleted.', workdir)
```

3.10 Files, Sockets, and similar Stateful Resources

Explicitly close files and sockets when done with them. This rule naturally extends to closeable resources that internally use sockets, such as database connections, and also other resources that need to be closed down in a similar fashion. To name only a few examples, this also includes mmap mappings, h5py File objects, and matplotlib.pyplot figure windows.

Leaving files, sockets or other such stateful objects open unnecessarily has many downsides:

- 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 such as moving or deleting them, or unmounting a filesystem.
- 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 raise exceptions, making the problem known sooner.

Furthermore, while files and sockets (and some similarly behaving resources) are automatically closed when the object is destructed, coupling the lifetime of the object to the state of the resource is poor practice:

- There are no guarantees as to when the runtime will actually invoke the `__del__` method. 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, e.g. in globals or exception tracebacks, may keep it around longer than intended.

Relying on finalizers to do automatic cleanup that has observable side effects has been rediscovered over and over again to lead to major problems, across many decades and multiple languages (see e.g. this article for Java).

The preferred way to manage files and similar resources 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("http://www.python.org/")) as front_page:
    for line in front_page:
        print(line)
```

In rare cases where context-based resource management is infeasible, code documentation must explain clearly how resource lifetime is managed.

3.11 TODO Comments

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

A `TODO` comment begins with the word `TODO` in all caps, a following colon, and a link to a resource that contains the context, ideally a bug reference. A bug reference is preferable because bugs are tracked and have follow-up comments. Follow this piece of context with an explanatory string introduced with a hyphen `-`. The purpose is to have a consistent `TODO` format that can be searched to find out how to get more details.

```
# TODO: crbug.com/192795 - Investigate cpufreq optimizations.
```

Old style, formerly recommended, but discouraged for use in new code:

```
# TODO(crbug.com/192795): Investigate cpufreq optimizations.
# TODO(yourusername): Use a "\*" here for concatenation operator.
```

Avoid adding TODOs that refer to an individual or team as the context:

```
# TODO: @yourusername - File an issue and use a '*' for repetition.
```

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.") that future code maintainers will comprehend. Issues are ideal for tracking this.

3.12 Imports formatting

Imports should be on separate lines; there are exceptions for `typing` and `collections.abc` imports.

E.g.:

```
Yes: from collections.abc import Mapping, Sequence
     import os
     import sys
     from typing import Any, NewType

No:  import os, sys
```

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 from most generic to least generic:

1. Python future import statements. For example:

   ```
   from __future__ import annotations
   ```

   See above for more information about those.

2. Python standard library imports. For example:

   ```
   import sys
   ```

3. third-party module or package imports. For example:

   ```
   import tensorflow as tf
   ```

4. Code repository sub-package imports. For example:

   ```
   from otherproject.ai import mind
   ```

5. **Deprecated:** application-specific imports that are part of the same top-level sub-package as this file. For example:

   ```
   from myproject.backend.hgwells import time_machine
   ```

   You may find older Google Python Style code doing this, but it is no longer required. **New code is encouraged not to bother with this.** Simply treat application-specific sub-package imports the same as other sub-package imports.

Within each grouping, imports should be sorted lexicographically, ignoring case, according to each module's full package path (the `path` in `from path import ...`). Code may optionally place a blank line between import sections.

```
import collections
import queue
import sys

from absl import app
from absl import flags
import bs4
import cryptography
import tensorflow as tf

from book.genres import scifi
from myproject.backend import huxley
from myproject.backend.hgwells import time_machine
from myproject.backend.state_machine import main_loop
from otherproject.ai import body
from otherproject.ai import mind
from otherproject.ai import soul

# Older style code may have these imports down here instead:
#from myproject.backend.hgwells import time_machine
#from myproject.backend.state_machine import main_loop
```

3.13 Statements

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)
```

3.14 Getters and Setters

Getter and setter functions (also called accessors and mutators) should be used when they provide a meaningful role or behavior for getting or setting a variable's value.

In particular, they should be used when getting or setting the variable is complex or the cost is significant, either currently or in a reasonable future.

If, for example, a pair of getters/setters simply read and write an internal attribute, the internal attribute should be made public instead. By comparison, if setting a variable means some state is invalidated or rebuilt, it should be a setter function. The function invocation hints that a potentially non-trivial operation is occurring. Alternatively, properties may be an option when simple logic is needed, or refactoring to no longer need getters and setters.

Getters and setters should follow the Naming guidelines, such as `get_foo()` and `set_foo()`.

If the past behavior allowed access through a property, do not bind the new getter/setter 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.

3.15 Naming

`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`, `query_proper_noun_for_thing`, `send_acronym_via_https`.

Function names, variable names, and filenames should be descriptive; avoid abbreviation. In particular, do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

Always use a `.py` filename extension. Never use dashes.

Names to Avoid

- single character names, except for specifically allowed cases:
  - counters or iterators (e.g. `i`, `j`, `k`, `v`, et al.)
  - `e` as an exception identifier in `try/except` statements.
  - `f` as a file handle in `with` statements
  - private type variables with no constraints (e.g. `_T = TypeVar("_T")`, `_P = ParamSpec("_P")`)

Please be mindful not to abuse single-character naming. Generally speaking, descriptiveness should be proportional to the name's scope of visibility. For example, `i` might be a fine name for 5-line code block but within multiple nested scopes, it is likely too vague.

- dashes (`-`) in any package/module name
- `__double_leading_and_trailing_underscore__` names (reserved by Python)
- offensive terms
- names that needlessly include the type of the variable (for example: `id_to_name_dict`)

Naming Conventions

- "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 (linters will flag protected member access). Note that it is okay for unit tests to access protected constants from the modules under test.
- Prepending a double underscore (`__` aka "dunder") to an instance variable or method effectively makes the variable or method private to its class (using name mangling); we discourage its use as it impacts readability and testability, and isn't *really* private. Prefer a single underscore.
- 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 some old 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`?")
- New *unit test* files follow PEP 8 compliant lower_with_under method names, for example, `test_<method_under_test>_<state>`. For consistency(*) with legacy modules that follow CapWords function names, underscores may appear in method names starting with `test` to separate logical components of the name. One possible pattern is `test<MethodUnderTest>_<state>`.

File Naming Python filenames must have a `.py` extension and must not contain dashes (`-`). This allows them to be imported and unittested. If you want an executable to be accessible without the extension, use a symbolic link or a simple bash wrapper containing `exec "$0.py" "$@"`.

Guidelines derived from Guido's

Mathematical Notation For mathematically heavy code, short variable names that would otherwise violate the style guide are preferred when they match established notation in a reference paper or algorithm. When doing so, reference the source of all naming conventions in a comment or docstring or, if the source is not accessible, clearly document the naming conventions. Prefer PEP8-compliant `descriptive_names` for public APIs, which are much more likely to be encountered out of context.

3.16 Main

In Python, `pydoc` as well as unit tests require modules to be importable. If a file is meant to be used as an executable, its main functionality should be in a `main()` function, and your code should always check `if __name__ == '__main__'` before executing your main program, so that it is not executed when the module is imported.

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

Table 1: Guidelines from Guido's Recommendations

When using absl, use `app.run`:

```
from absl import app
...

def main(argv: Sequence[str]):
    # process non-flag arguments
    ...

if __name__ == '__main__':
    app.run(main)
```

Otherwise, use:

```
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.

3.17 Function length

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on function length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

3.18 Type Annotations

General Rules

- Familiarize yourself with PEP-484.
- Annotating `self` or `cls` is generally not necessary. `Self` can be used if it is necessary for proper type information, e.g.

```
from typing import Self

class BaseClass:
  @classmethod
  def create(cls) -> Self:
    ...

  def difference(self, other: Self) -> float:
    ...
```

- Similarly, don't feel compelled to annotate the return value of `__init__` (where `None` is the only valid option).
- If any other variable or a returned type should not be expressed, use `Any`.
- You are not required to annotate all the functions in a module.
  - At least annotate your public APIs.
  - Use judgment to get to a good balance between safety and clarity on the one hand, and flexibility on the other.
  - Annotate code that is prone to type-related errors (previous bugs or complexity).
  - Annotate code that is hard to understand.
  - Annotate code as it becomes stable from a types perspective. In many cases, you can annotate all the functions in mature code without losing too much flexibility.

Line Breaking Try to follow the existing indentation rules.

After annotating, many function signatures will become "one parameter per line". To ensure the return type is also given its own line, a comma can be placed after the last parameter.

```
def my_method(
    self,
    first_var: int,
    second_var: Foo,
    third_var: Bar | None,
) -> int:
  ...
```

Always prefer breaking between variables, and not, for example, between variable names and type annotations. However, if everything fits on the same line, go for it.

```
def my_method(self, first_var: int) -> int:
  ...
```

If the combination of the function name, the last parameter, and the return type is too long, indent by 4 in a new line. When using line breaks, prefer putting each parameter and the return type on their own lines and

aligning the closing parenthesis with the `def`:

```
Yes:
def my_method(
    self,
    other_arg: MyLongType | None,
) -> tuple[MyLongType1, MyLongType1]:
  ...
```

Optionally, the return type may be put on the same line as the last parameter:

```
Okay:
def my_method(
    self,
    first_var: int,
    second_var: int) -> dict[OtherLongType, MyLongType]:
  ...
```

`pylint` allows you to move the closing parenthesis to a new line and align with the opening one, but this is less readable.

```
No:
def my_method(self,
              other_arg: MyLongType | None,
             ) -> dict[OtherLongType, MyLongType]:
  ...
```

As in the examples above, prefer not to break types. However, sometimes they are too long to be on a single line (try to keep sub-types unbroken).

```
def my_method(
    self,
    first_var: tuple[list[MyLongType1],
                     list[MyLongType2]],
    second_var: list[dict[
        MyLongType3, MyLongType4]],
) -> None:
  ...
```

If a single name and type is too long, consider using an alias for the type. The last resort is to break after the colon and indent by 4.

```
Yes:
def my_function(
    long_variable_name:
        long_module_name.LongTypeName,
) -> None:
  ...
```

```
No:
def my_function(
    long_variable_name: long_module_name.
        LongTypeName,
) -> None:
  ...
```

Forward Declarations If you need to use a class name (from the same module) that is not yet defined – for example, if you need the class name inside the declaration of that class, or if you use a class that is defined

later in the code – either use `from __future__ import annotations` or use a string for the class name.

```
Yes:
from __future__ import annotations

class MyClass:
  def __init__(self, stack: Sequence[MyClass], item: OtherClass) -> None:

class OtherClass:
  ...
```

```
Yes:
class MyClass:
  def __init__(self, stack: Sequence['MyClass'], item: 'OtherClass') -> None:

class OtherClass:
  ...
```

Default Values As per PEP-008, use spaces around the `=` *only* for arguments that have both a type annotation and a default value.

```
Yes:
def func(a: int = 0) -> int:
  ...
```

```
No:
def func(a:int=0) -> int:
  ...
```

NoneType In the Python type system, `NoneType` is a "first class" type, and for typing purposes, `None` is an alias for `NoneType`. If an argument can be `None`, it has to be declared! You can use `|` union type expressions (recommended in new Python 3.10+ code), or the older `Optional` and `Union` syntaxes.

Use explicit `X | None` instead of implicit. Earlier versions of PEP 484 allowed `a: str = None` to be interpreted as `a: str | None = None`, but that is no longer the preferred behavior.

```
Yes:
def modern_or_union(a: str | int | None, b: str | None = None) -> str:
  ...
def union_optional(a: Union[str, int, None], b: Optional[str] = None) -> str:
  ...
```

```
No:
def nullable_union(a: Union[None, str]) -> str:
  ...
def implicit_optional(a: str = None) -> str:
  ...
```

Type Aliases You can declare aliases of complex types. The name of an alias should be CapWorded. If the alias is used only in this module, it should be _Private.

Note that the `: TypeAlias` annotation is only supported in versions 3.10+.

```
from typing import TypeAlias

_LossAndGradient: TypeAlias = tuple[tf.Tensor, tf.Tensor]
ComplexTFMap: TypeAlias = Mapping[str, _LossAndGradient]
```

Ignoring Types You can disable type checking on a line with the special comment `# type: ignore`.

`pytype` has a disable option for specific errors (similar to lint):

```
# pytype: disable=attribute-error
```

Typing Variables *Annotated Assignments* : If an internal variable has a type that is hard or impossible to infer, specify its type with an annotated assignment - use a colon and type between the variable name and value (the same as is done with function arguments that have a default value):

```python
a: Foo = SomeUndecoratedFunction()
```

Type Comments : Though you may see them remaining in the codebase (they were necessary before Python 3.6), do not add any more uses of a `# type: <type name>` comment on the end of the line:

```python
a = SomeUndecoratedFunction()  # type: Foo
```

Tuples vs Lists Typed lists can only contain objects of a single type. Typed tuples can either have a single repeated type or a set number of elements with different types. The latter is commonly used as the return type from a function.

```
a: list[int] = [1, 2, 3]
b: tuple[int, ...] = (1, 2, 3)
c: tuple[int, str, float] = (1, "2", 3.5)
```

3Type variables The Python type system has generics. A type variable, such as `TypeVar` and `ParamSpec`, is a common way to use them.

Example:

```
from collections.abc import Callable
from typing import ParamSpec, TypeVar
_P = ParamSpec("_P")
_T = TypeVar("_T")
...
def next(l: list[_T]) -> _T:
  return l.pop()

def print_when_called(f: Callable[_P, _T]) -> Callable[_P, _T]:
  def inner(*args: _P.args, **kwargs: _P.kwargs) -> _T:
    print("Function was called")
    return f(*args, **kwargs)
  return inner
```

A `TypeVar` can be constrained:

```
AddableType = TypeVar("AddableType", int, float, str)
def add(a: AddableType, b: AddableType) -> AddableType:
  return a + b
```

A common predefined type variable in the `typing` module is `AnyStr`. Use it for multiple annotations that can

be `bytes` or `str` and must all be the same type.

```
from typing import AnyStr
def check_length(x: AnyStr) -> AnyStr:
  if len(x) <= 42:
    return x
  raise ValueError()
```

A type variable must have a descriptive name, unless it meets all of the following criteria:

- not externally visible
- not constrained

```
Yes:
  _T = TypeVar("_T")
  _P = ParamSpec("_P")
  AddableType = TypeVar("AddableType", int, float, str)
  AnyFunction = TypeVar("AnyFunction", bound=Callable)
```

```
No:
  T = TypeVar("T")
  P = ParamSpec("P")
  _T = TypeVar("_T", int, float, str)
  _F = TypeVar("_F", bound=Callable)
```

String types

> Do not use `typing.Text` in new code. It's only for Python 2/3 compatibility.

Use `str` for string/text data. For code that deals with binary data, use `bytes`.

```
def deals_with_text_data(x: str) -> str:
  ...
def deals_with_binary_data(x: bytes) -> bytes:
  ...
```

If all the string types of a function are always the same, for example if the return type is the same as the argument type in the code above, use AnyStr.

Imports For Typing For symbols (including types, functions, and constants) from the `typing` or `collections.abc` modules used to support static analysis and type checking, always import the symbol itself. This keeps common annotations more concise and matches typing practices used around the world. You are explicitly allowed to import multiple specific symbols on one line from the `typing` and `collections.abc` modules. For example:

```
from collections.abc import Mapping, Sequence
from typing import Any, Generic, cast, TYPE_CHECKING
```

Given that this way of importing adds items to the local namespace, names in `typing` or `collections.abc` should be treated similarly to keywords, and not be defined in your Python code, typed or not. If there is a collision between a type and an existing name in a module, import it using `import x as y`.

```
from typing import Any as AnyType
```

Prefer to use built-in types as annotations where available. Python supports type annotations using parametric container types via PEP-585, introduced in Python 3.9.

```
def generate_foo_scores(foo: set[str]) -> list[float]:
  ...
```

Conditional Imports Use conditional imports only in exceptional cases where the additional imports needed for type checking must be avoided at runtime. This pattern is discouraged; alternatives such as refactoring the code to allow top-level imports should be preferred.

Imports that are needed only for type annotations can be placed within an `if TYPE_CHECKING:` block.

- Conditionally imported types need to be referenced as strings, to be forward compatible with Python 3.6 where the annotation expressions are actually evaluated.
- Only entities that are used solely for typing should be defined here; this includes aliases. Otherwise it will be a runtime error, as the module will not be imported at runtime.
- The block should be right after all the normal imports.
- There should be no empty lines in the typing imports list.
- Sort this list as if it were a regular imports list.

```
import typing
if typing.TYPE_CHECKING:
  import sketch
def f(x: "sketch.Sketch"): ...
```

Circular Dependencies Circular dependencies that are caused by typing are code smells. Such code is a good candidate for refactoring. Although technically it is possible to keep circular dependencies, various build systems will not let you do so because each module has to depend on the other.

Replace modules that create circular dependency imports with `Any`. Set an alias with a meaningful name, and use the real type name from this module (any attribute of `Any` is `Any`). Alias definitions should be separated from the last import by one line.

```
from typing import Any

some_mod = Any  # some_mod.py imports this module.
...

def my_method(self, var: "some_mod.SomeType") -> None:
  ...
```

Generics When annotating, prefer to specify type parameters for generic types; otherwise, the generics' parameters will be assumed to be `Any`.

```
# Yes:
def get_names(employee_ids: Sequence[int]) -> Mapping[int, str]:
  ...

# No:
# This is interpreted as get_names(employee_ids: Sequence[Any]) -> Mapping[Any, Any]
def get_names(employee_ids: Sequence) -> Mapping:
  ...
```

If the best type parameter for a generic is `Any`, make it explicit, but remember that in many cases `TypeVar` might be more appropriate:

```
# No:
def get_names(employee_ids: Sequence[Any]) -> Mapping[Any, str]:
  """Returns a mapping from employee ID to employee name for given IDs."""

# Yes:
_T = TypeVar('_T')
def get_names(employee_ids: Sequence[_T]) -> Mapping[_T, str]:
  """Returns a mapping from employee ID to employee name for given IDs."""
```

4 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 `_idx` suffixes in index variable names, 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.

However, there are limits to consistency. It applies more heavily locally and on choices unspecified by the global style. Consistency should not generally be used as a justification to do things in an old style without considering the benefits of the new style, or the tendency of the codebase to converge on newer styles over time.