Google Python 风格指南

2025 年 9 月 5 日

目录

1 背景

Python 是 Google 使用的主要动态语言。本风格指南列出了 Python 程序的**应该做和不应该做**。

为了帮助您正确格式化代码，我们创建了一个 Vim 的设置文件[1]。对于 Emacs，默认设置应该就够用了。

许多团队使用 Black[2] 或 Pyink[3] 自动格式化器来避免争论格式化问题。

2 Python 语言规则

2.1 Lint

使用本项目的 pylintrc[4]，并运行 `pylint` 检查您的代码。

2.1.1 定义

`pylint` 是一个用于查找 Python 源代码中 bug 和风格问题的工具。它会发现通常由非动态的语言如 C 和 C++ 编译器捕获的问题。由于 Python 的动态特性，有些警告可能不正确；然而，虚假警告应该很少见。

2.1.2 优点

捕获容易遗漏的错误，如拼写错误、使用变量前未赋值等。

2.1.3 缺点

`pylint` 并不完美。要利用它，有时我们需要绕过它、抑制其警告或修复它。

[1] https://github.com/google/styleguide/blob/gh-pages/google_python_style.vim
[2] https://github.com/psf/black
[3] https://github.com/google/pyink
[4] https://github.com/google/styleguide/blob/gh-pages/pylintrc

2.1.4 决定

确保运行 pylint 检查您的代码。

如果警告不合适，请抑制它们，以免隐藏其他问题。您可以在行级注释中设置：

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

pylint 警告每个都由符号名称（empty-docstring）标识。Google 特定的警告以 g- 开头。

如果抑制的原因从符号名称中不清楚，请添加解释。

以这种方式抑制的好处是我们可以轻松搜索抑制并重新审视它们。

您可以通过以下方式获取 pylint 警告列表：

```
pylint --list-msgs
```

要获取特定消息的更多信息，请使用：

```
pylint --help-msg=invalid-name
```

优先使用 pylint: disable 而非已弃用的旧形式 pylint: disable-msg。

可以通过在函数开头删除变量来抑制未使用参数警告。请始终包含解释为什么删除它的注释。“Unused.” 就足够了。例如：

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

抑制此警告的其他常见形式包括使用 ‘_’ 作为未使用参数的标识符，或在参数名称前添加 ‘unused_’，或将它们分配给 ‘_’。这些形式允许但不再鼓励。这些会破坏按名称传递参数的调用者，并且不强制参数实际上未被使用。

2.2 导入

仅使用 import 语句导入包和模块，而非单个类型、类或函数。

2.2.1 定义

从一个模块共享代码到另一个模块的重用机制。

2.2.2 优点

命名空间管理约定简单。每个标识符的来源以一致的方式指示；`x.Obj` 表示对象 `Obj` 在模块 `x` 中定义。

2.2.3 缺点

模块名称仍可能冲突。有些模块名称过于冗长。

2.2.4 决定

- 使用 `import x` 导入包和模块。
- 使用 `from x import y`，其中 `x` 是包前缀，`y` 是无前缀的模块名称。
- 在以下任何情况下使用 `from x import y as z`：
  - 需要导入两个名为 `y` 的模块。
  - `y` 与当前模块中定义的顶级名称冲突。
  - `y` 与公共 API 中常见的参数名称冲突（例如，`features`）。
  - `y` 是一个过于冗长的名称。
  - `y` 在您的代码上下文中过于通用(例如,`from storage.file_system import options as fs_options`)。
- 仅当 `z` 是标准缩写时使用 `import y as z`（例如，`import numpy as np`）。

例如，模块 `sound.effects.echo` 可以如下导入：

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

在导入中不要使用相对名称。即使模块在同一包中，也要使用完整的包名称。这有助于防止意外导入一个包两次。

2.2.4.1 豁免 此规则的豁免：

- 来自以下模块的符号用于支持静态分析和类型检查：

  - `typing` 模块（见用于类型的导入）
  - `collections.abc` 模块（见用于类型的导入）
  - `typing_extensions` 模块[5]
- 来自 `six.moves` 模块[6] 的重定向。

2.3 包

使用模块的完整路径位置导入每个模块。

2.3.1 优点

避免模块名称冲突或由于模块搜索路径不是作者预期的那样而导致的错误导入。更容易找到模块。

2.3.2 缺点

使部署代码更难，因为您必须复制包层次结构。使用现代部署机制并不是真正的问题。

2.3.3 决定

所有新代码都应通过其完整包名称导入每个模块。

导入应如下：

正确：

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

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

正确：

```
# Reference flags in code with just the module name (common).
from absl import flags
```

[5] https://github.com/python/typing_extensions/blob/main/README.md

[6] https://six.readthedocs.io/#module-six.moves

```
from doctor.who import jodie

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

(假设此文件位于 doctor/who/ 中，其中 jodie.py 也存在)

错误：

```
# 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
```

主二进制文件所在的目录不应假设在 `sys.path` 中，尽管在某些环境中会发生这种情况。既然如此，代码应假设 `import jodie` 指的是名为 `jodie` 的第三方或顶级包，而不是本地的 `jodie.py`。

2.4 异常

允许使用异常，但必须小心使用。

2.4.1 定义

异常是一种跳出正常控制流的手段，用于处理错误或其他异常情况。

2.4.2 优点

正常操作代码的控制流不会因错误处理（error-handling）代码而变得杂乱。它还允许当某些条件发生时，控制流跳过多个帧（frames），例如，从第 N 层嵌套函数中一步返回，而不是必须通过它们传递错误代码。

2.4.3 缺点

可能导致控制流令人困惑。在调用库时容易遗漏错误情况。

2.4.4 决定

异常必须遵循某些条件：

- 当合适时，使用内置异常类。例如，当违反前提条件时（如验证函数参数时可能发生的情况），引发 `ValueError` 以指示编程错误。
- 不要使用 `assert` 语句代替条件或验证前提条件。它们不得对应用程序逻辑至关重要。试金石是 `assert` 可以被移除而不破坏代码。`assert` 条件不能保证被求值[7]。对于基于 pytest[8] 的测试，`assert` 是可以的，并且期望用于验证期望。例如：

  正确：

```
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
```

  错误：

```
def connect_to_next_port(self, minimum: int) -> int:
```

[7] https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

[8] https://pytest.org

```
    """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
```

- 库或包可能定义自己的异常。当这样做时，它们必须从现有异常类继承。异常名称应以 `Error` 结尾，并且不应引入重复（`foo.FooError`）。
- 永远不要使用捕获所有异常的（catch-all）`except:` 语句，或捕获 `Exception` 或 `StandardError`，除非您是

  – 重新引发异常，或者
  – 在程序中创建一个隔离点，其中异常不传播但被记录和抑制，例如，保护线程免于崩溃，通过守护其最外层块。

  Python 在这方面非常宽容，`except:` 真的会捕获一切，包括拼写错误的名称、`sys.exit()` 调用、Ctrl+C 中断、unittest 失败以及您根本不想捕获的其他各种异常。
- 最小化 `try/except` 块中的代码量。`try` 的主体越大，您没想到会引发异常的代码行引发异常的可能性越大。在这些情况下，`try/except` 块会隐藏真正的错误。
- 使用 `finally` 子句无论 `try` 块中是否引发异常都执行代码。这通常用于清理，例如关闭文件。

2.5 可变全局状态

避免可变全局状态。

2.5.1 定义

模块级值或类属性，可以在程序执行期间被修改。

2.5.2 优点

偶尔有用。

2.5.3 缺点

- 破坏封装：这样的设计会使实现有效目标变得困难。例如，如果全局状态用于管理数据库连接，那么同时连接到两个不同的数据库（例如用于计算迁移期间的差异）会变得困难。类似问题很容易在全局注册表中出现。
- 有可能在导入期间改变模块行为，因为在模块首次导入时会分配全局变量。

2.5.4 决定

避免可变全局状态。

在极少数情况下使用全局状态是合理的，可变全局实体应在模块级或作为类属性声明，并通过在其名称前添加 `_` 使其内部化。如果需要，通过公共函数或类方法外部访问可变全局状态。请参阅下面的命名。请在注释或从注释链接的文档中解释使用可变全局状态的设计原因。

模块级常量允许且鼓励。例如：内部使用常量的 `_MAX_HOLY_HANDGRENADE_COUNT = 3` 或公共 API 常量的 `SIR_LANCELOTS_FAVORITE_COLOR = "blue"`。常量必须使用全大写并带下划线命名。请参阅下面的命名。

2.6 嵌套/局部/内部类和函数

当用于闭包局部变量时，嵌套局部函数或类是可以的。内部类是可以的。

2.6.1 定义

可以在方法、函数或类内部定义类。可以在方法或函数内部定义函数。嵌套函数对封闭作用域中定义的变量具有只读访问权限[9]。

2.6.2 优点

允许定义仅在非常有限的作用域内使用的实用类和函数。非常 ADT[10] 风格。通常用于实现装饰器。

2.6.3 缺点

嵌套函数和类无法直接测试。嵌套会使外部函数更长且可读性更差。

2.6.4 决定

可以使用，但有一些注意事项。除了闭包 `self` 或 `cls` 以外的局部值之外，避免嵌套函数或类。不要仅为了从模块用户隐藏而嵌套函数。相反，在模块级为其名称添加 `_` 前缀，以便测试仍能访问它。

2.7 推导式 & 生成器表达式

对于简单情况可以使用。

2.7.1 定义

列表、字典和集合推导式以及生成器表达式提供了一种简洁高效的方式来创建容器类型和迭代器，而无需使用传统的循环、`map()`、`filter()` 或 `lambda`。

[9] (译者注) 这部分在 Python 中是基本正确的，但需要一些补充说明。在 Python 中，嵌套函数（nested functions）可以读取（read）定义在外部作用域（enclosing scope）中的变量，但默认情况下不能直接修改这些变量。如果需要修改外部作用域的变量，必须使用 nonlocal 关键字（Python 3 引入）来声明。

[10] https://en.wikipedia.org/wiki/Abstract_data_type

2.7.2 优点

简单的推导式可以比其他字典、列表或集合创建技术更清晰、更简单。生成器表达式可以非常高效，因为它们避免创建整个列表。

2.7.3 缺点

复杂的推导式或生成器表达式可能难以阅读。

2.7.4 决定

允许使用推导式，但是不允许多个 `for` 子句或过滤表达式。以可读性为优化目标，而不是简洁性。

正确:

```
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( ):
  for y in range( ):
    if x * y > :
      result.append((x, y))

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

return (x** for x in range( ))
```

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

错误：

```
result = [(x, y) for x in range(5) 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 默认迭代器和运算符

对于支持它们的类型，如列表、字典和文件，使用默认迭代器和运算符。

2.8.1 定义

容器类型，如字典和列表，定义了默认迭代器和成员测试运算符（“`in`”和“`not in`”）。

2.8.2 优点

默认迭代器和运算符简单且高效。它们直接表达操作，而无额外方法调用。使用默认运算符的函数是通用的。它可以用于任何支持该操作的类型。

2.8.3 缺点

通过阅读方法名称无法判断对象的类型（除非变量有类型注解）。这也是一个优点。

2.8.4 决定

对于支持它们的类型，如列表、字典和文件，使用默认迭代器和运算符。内置类型也定义了迭代器方法。优先使用这些方法而不是返回列表的方法，除非您不应在迭代容器时修改它。

正确:

```
for key in adict: ...
if obj in alist: ...
for line in afile: ...
for k, v in adict.items(): ...
```

错误:

```
for key in adict.keys(): ...
for line in afile.readlines(): ...
```

2.9 生成器

根据需要使用生成器。

2.9.1 定义

生成器函数返回一个迭代器，每次执行 `yield` 语句时产生一个值。在产生值后，生成器函数的运行时状态被挂起，直到需要下一个值。

2.9.2 优点

更简单的代码，因为每次调用都会保留局部变量的状态和控制流。生成器比一次性创建整个值列表的函数使用更少的内存。

2.9.3 缺点

生成器中的局部变量在生成器耗尽或本身被垃圾回收之前不会被垃圾回收。

2.9.4 决定

可以。在生成器函数的文档字符串中使用 “Yields:” 而非 “Returns:”。

如果生成器管理昂贵的资源，请确保强制清理。

进行清理的好方法是使用上下文管理器（context manager）包装生成器 PEP 0533[11]。

[11] https://peps.python.org/pep-0533/

2.10 Lambda 函数

对于单行可以使用。优先使用生成器表达式而非带有 `lambda` 的 `map()` 或 `filter()`。

2.10.1 定义

Lambda 在表达式中定义匿名函数，而不是语句。

2.10.2 优点

方便。

2.10.3 缺点

比局部函数更难阅读和调试。没有名称意味着栈跟踪更难理解。表现力有限，因为函数只能包含表达式。

2.10.4 决定

允许使用 Lambda。如果 lambda 函数内的代码跨越多行或长于 60-80 个字符，可能最好将其定义为常规嵌套函数 (见词法作用域)。

对于常见操作如乘法，使用 `operator` 模块中的函数而不是 lambda 函数。例如，优先使用 `operator.mul` 而非 `lambda x, y: x * y`。

2.11 条件表达式

对于简单情况可以使用。

2.11.1 定义

条件表达式（有时称为“三元运算符”）是为 if 语句提供更短语法的机制。例如：`x = 1 if cond else 2`。

2.11.2 优点

比 `if` 语句更短、更方便。

2.11.3 缺点

可能比 `if` 语句更难阅读。如果表达式很长，条件可能难以定位。

2.11.4 决定

对于简单情况可以使用。每部分必须在一行上：true-expression, if-expression, else-expression。当事情变得更复杂时，使用完整的 `if` 语句。

正确：

```
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')
```

错误：

```
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 默认参数值

在大多数情况下可以使用。

2.12.1 定义

您可以在函数参数列表末尾指定变量的值，例如，`def foo(a, b=0):`。如果仅用一个参数调用 `foo`，则 `b` 设置为 0。如果用两个参数调用，则 `b` 具有第二个参数的值。

2.12.2 优点

通常您有一个使用大量默认值的函数，但偶尔您想覆盖默认值。默认参数值提供了一种简单的方法来实现这一点，而无需为罕见的例外定义许多函数。由于 Python 不支持重载方法/函数，默认参数是一种 “模拟” 重载行为的简单方式。

2.12.3 缺点

默认参数在模块加载时求值一次。如果参数是可变对象，如列表或字典，这可能会导致问题。如果函数修改对象（例如，通过向列表追加项），默认值会被修改。

2.12.4 决定

可以使用，但有以下注意事项:

不要在函数或方法定义中使用可变对象作为默认值。

正确:

```
def foo(a, b=None):
    if b is None:
        b = []
def foo(a, b: Sequence | None = None):
    if b is None:
        b = []
def foo(a, b: Sequence = ()):  # Empty tuple OK since tuples are
   immutable.
    ...
```

错误:

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

def foo(a, b=[]):
    ...
```

```
def foo(a, b=time.time()):  # Is `b` supposed to represent when this
    module was loaded?
    ...
def foo(a, b=_FOO.value):  # sys.argv has not yet been parsed...
    ...
def foo(a, b: Mapping = {}):  # Could still get passed to unchecked
   code.
    ...
```

2.13 属性

当需要为获取或设置属性进行琐碎计算或逻辑时，可以使用属性。属性实现必须匹配常规属性访问的一般期望：廉价、直截、符合预期。

2.13.1 定义

一种将方法调用包装为标准属性访问以获取和设置属性的方式。

2.13.2 优点

- 允许使用属性访问和赋值 API 而非 Getters 和 Setters 方法调用。
- 可用于使属性只读。
- 允许延迟计算。
- 提供一种在内部演化时保持类公共接口的方式，与类用户独立。

2.13.3 缺点

- 可以像运算符重载一样隐藏副作用。
- 对于子类可能令人困惑。

2.13.4 决定

允许使用属性，但像运算符重载一样，仅在必要时使用，并匹配典型属性访问的期望；否则遵循 Getters 和 Setters 规则。

例如，使用属性简单地获取和设置内部属性是不允许的：没有计算发生，因此属性是不必要的（改为使用公有属性，见Getters 和 Setters）。相比之下，使用属性控制属性访问或计算琐碎派生值是允许的：逻辑简单且无意外。

应使用 `@property` （见函数和方法装饰器）创建属性。手动实现属性描述符被视为强大功能。

属性的继承可能不明显。不要使用属性实现子类可能想要重写和扩展的计算。

2.14 True/False 求值

尽可能使用 “隐式” false（有一些注意事项）。

2.14.1 定义

在布尔上下文中，Python 将某些值评估为 `False`。一个快速的 “经验法则” 是所有 “空” 值都被视为 false，因此 `，None，[]，{}，''` 在布尔上下文中都求值为 false。

2.14.2 优点

使用 Python 布尔值的条件更容易阅读且不易出错。在大多数情况下，它们也更快。

2.14.3 缺点

对 C/C++ 开发人员可能看起来奇怪。

2.14.4 决定

尽可能使用 “隐式” `false`，例如，`if foo:` 而非 `if foo != []:`。但有一些注意事项需要牢记：

- 始终使用 `if foo is None:`（或 `is not None`）检查 `None` 值。例如，当测试默认值为 `None` 的变量或参数是否设置为其他值时。其他值可能在布尔上下文中为 `false`！
- 永远不要使用 == 将布尔变量与 `False` 比较。改为使用 `if not x:`。如果需要区分 `False` 和 `None`，则链式表达式，如 `if not x and x is not None:`。

- 对于序列（字符串、列表、元组），使用空序列为 `false` 的事实，因此 `if seq:` 和 `if not seq:` 分别优于 `if len(seq):` 和 `if not len(seq):`。
- 处理整数时，隐式 `false` 可能涉及比好处更多的风险（即，意外地将 `None` 处理为 0）。您可以将已知为整数的值（且不是 `len()` 的结果）与整数 0 比较。

  正确：

```
if not users:
    print('no users')

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

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

  错误：

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

if not i % 10:
    self.handle_multiple_of_ten()

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

- 注意，`'0'`（即，字符串形式的 0）求值为 `true` 。
- 注意，Numpy 数组在隐式布尔上下文中可能引发异常。在测试 `np.array` 的空时，优先使用 `.size` 属性（例如 `if not users.size`）。

2.15 词法作用域

可以使用。

2.15.1 定义

嵌套的 Python 函数可以引用封闭函数中定义的变量，但不能为其赋值。变量绑定使用词法作用域解析，即基于静态程序文本。对块中名称的任何赋值都会导致 Python

将对该名称的所有引用视为局部变量，即使使用在赋值之前。如果发生全局声明，则名称被视为全局变量。

此功能的示例是：

```
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
```

2.15.2 优点

通常会导致更清晰、更优雅的代码。尤其是对有经验的 Lisp 和 Scheme（以及 Haskell 和 ML 等）程序员来说很舒适。

2.15.3 缺点

可能导致令人困惑的 bug，例如基于 PEP 0227[12] 的此示例：

```
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()
```

所以 `foo([1, 2, 3])` 将打印 `1 2 3 3`，而不是 `1 2 3 4`。

2.15.4 决定

可以使用。

[12] https://peps.python.org/pep-0227/

2.16 函数和方法装饰器

当有明显优势时，谨慎使用装饰器。避免 `staticmethod`，并限制使用 `classmethod`。

2.16.1 定义

函数和方法的装饰器[13]（又称 “@ 符号”）。一个常见的装饰器是 `@property`，用于将普通方法转换为动态计算的属性。然而，装饰器语法也允许用户定义装饰器。具体来说，对于某些函数 `my_decorator`，这：

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

等价于：

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

2.16.2 优点

优雅地指定对方法的某些转换；转换可能消除一些重复代码、强制不变性等。

2.16.3 缺点

装饰器可以对函数的参数或返回值执行任意操作，导致意外的隐式行为。此外，装饰器在对象定义时执行。对于模块级对象（类、模块函数等），这发生在导入时。装饰器代码中的失败几乎不可能恢复。

2.16.4 决定

当有明显优势时，谨慎使用装饰器。装饰器应遵循与函数相同的导入和命名指南。装饰器文档字符串应清楚说明函数是装饰器。为装饰器编写单元测试。

[13] https://docs.python.org/3/glossary.html#term-decorator

避免装饰器本身中的外部依赖（例如，不要依赖文件、套接字、数据库连接等），因为在装饰器运行时（导入时，可能来自 `pydoc` 或其他工具）它们可能不可用。用有效参数调用装饰器应（尽可能）保证在所有情况下成功。

装饰器是 “顶级代码” 的特殊情况 - 有关更多讨论，请参阅 [主程序]。

除非被迫整合现有库中定义的 API，否则不要使用 `staticmethod`。改为编写模块级函数。

仅当编写命名构造函数，或修改必要全局状态（如进程范围缓存）的类特定例程时，使用 `classmethod`。

2.17 线程

不要依赖内置类型的原子性。

虽然 Python 的内置数据类型如字典似乎具有原子操作，但存在角案例，它们不是原子性的（例如，如果 `__hash__` 或 `__eq__` 实现为 Python 方法），并且不应依赖它们的原子性。也不应依赖原子变量赋值（因为这反过来依赖字典）。

使用 `queue` 模块的 `Queue` 数据类型作为线程之间通信数据的首选方式。否则，使用 `threading` 模块及其锁原语。优先使用条件变量和 `threading.Condition` 而非使用低级锁。

2.18 强大功能

避免这些功能。

2.18.1 定义

Python 是一种极其灵活的语言，提供了许多高级功能，如自定义元类、访问字节码、动态编译、动态继承、对象重新父化、导入黑客、反射（例如 `getattr()` 的某些使用）、修改系统内部、实现自定义清理的 `__del__` 方法等。

2.18.2 优点

这些是强大的语言功能。它们可以使您的代码更紧凑。

2.18.3 缺点

当它们不是绝对必要时，使用这些 “酷” 功能非常诱人。阅读、理解和调试使用底层不寻常功能的代码更难。一开始（对原作者）似乎不是这样，但当重新审视代码时，它往往比更长但更直接的代码更难。

2.18.4 决定

在您的代码中避免这些功能。

内部使用这些功能的标准库模块和类是可以使用的（例如，`abc.ABCMeta`、`dataclasses` 和 `enum`）。

2.19 现代 Python：`from __future__ import`

新语言版本语义变化可能通过特殊的 future 导入在每个文件的基础上在早期运行时启用。

2.19.1 定义

能够通过 `from __future__ import` 语句开启一些更现代的功能，允许在预期未来 Python 版本中提前使用功能。

2.19.2 优点

这已被证明使运行时版本升级更平滑，因为可以在每个文件的基础上进行更改，同时声明兼容性并防止这些文件中的回归。现代代码更易维护，因为它不太可能积累在未来运行时升级中会有问题的技术债务。

2.19.3 缺点

这样的代码可能无法在引入所需 future 语句之前的非常旧的解释器版本上运行。在支持极其广泛环境的项目中，这种需求更常见。

2.19.4 决定

2.19.4.1 `from __future__ import` 鼓励使用 `from __future__ import` 语句。它允许给定源文件今天开始使用更现代的 Python 语法功能。一旦您不再需要在隐藏在 `__future__` 导入后面的功能版本上运行，请随意删除这些行。

在代码上运行版本如 3.5 而非 >= 3.7 时，导入：

```
from __future__ import generator_stop
```

有关更多信息，请阅读 Python future 语句定义文档[14]。

请不要删除这些导入，直到您确信代码仅在足够现代的环境中使用。即使您今天没有在代码中使用特定 future 导入启用的功能，将其保留在文件中可以防止代码的后续修改无意中依赖旧行为。

根据需要使用其他 `from __future__ import` 语句。

2.20 类型注解代码

您可以使用类型提示（type hints）[15]注解 Python 代码。在构建时使用类型检查工具如 pytype[16] 对代码进行类型检查。在大多数情况下，当可行时，类型注解在源文件中。对于第三方或扩展模块，注解可以在存根 `.pyi` 文件[17]中。

2.20.1 定义

类型注解（或“类型提示”）用于函数或方法参数和返回值：

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

您也可以使用类似语法声明变量的类型：

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

[14] https://docs.python.org/3/library/__future__.html
[15] https://docs.python.org/3/library/typing.html
[16] https://github.com/google/pytype
[17] https://peps.python.org/pep-0484/#stub-files

2.20.2 优点

类型注解提高了代码的可读性和可维护性。类型检查器可以捕获许多运行时错误，而无需编写单元测试。

2.20.3 缺点

您必须保持类型注解最新。类型检查可能会减慢您的代码（尽管 pytype 有增量检查）。您的团队可能尚未习惯它们。

2.20.4 决定

强烈建议在更新代码时启用 Python 类型分析。在添加或修改公共 API 时，请包含类型注解，并在构建系统中启用通过 pytype 进行检查。由于静态分析在 Python 中相对较新，我们认识到一些不必要的副作用（例如错误推断的类型）可能会阻碍某些项目采用类型注解。在这些情况下，鼓励作者在 BUILD 文件或代码本身中添加注释，包含 TODO 或指向描述当前阻止类型注解采用的问题的 bug 链接。

3 Python 风格规则

3.1 分号

不要在语句末尾放置分号，也不要使用分号将两个语句放在同一行。

3.2 行长度

最大行长度为 80 个字符。

例外：

- 长导入模块语句。
- 注释中的 URL、路径名和长标志。
- 长字符串文字（参见字符串）。
  - Pylint 禁用注释。（例如：`# pylint: disable=invalid-name`）

不要使用反斜杠来显式续行[18]。

使用 Python 的 () [] {} 来隐式续行[19]。如有必要，您可以用添加额外的 () 包裹表达式。

注意：此规则不禁止在字符串中使用反斜杠转义的换行符（见字符串）。

正确：

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

正确：

```
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)
```

错误：

```
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)
```

当字符串字面量无法在一行内适应时，使用括号进行隐式行连接：

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

[18] https://docs.python.org/3/reference/lexical_analysis.html#explicit-line-joining

[19] https://docs.python.org/3/reference/lexical_analysis.html#implicit-line-joining

优先在尽可能高的语法级别断行。如果必须两次断行，两次断行应保持在同一语法级别。

正确：

```
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()
```

错误：

```
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()
```

在注释中，如有需要，将长 URL 单独放在一行：

正确：

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

错误：

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

注意上述行延续示例中元素的缩进；有关缩进的解释，请参见缩进部分。

文档字符串（Docstring）摘要行必须保持在 80 个字符的限制内（见文档字符串）。

在所有其他情况下，如果一行超过 80 个字符，并且 Black 或 Pyink 自动格式化工具无法将行限制在该范围内，则允许该行超过此最大值。鼓励作者在合理的情况下按照上述说明手动断行。

3.3 括号

谨慎使用括号。

不要在返回语句或条件语句中使用它们，除非使用括号来实现行续接或指示优先级。例外情况是元组语法（`return x, y`），其中括号是必需的；和 f-字符串，其中括号用于格式化或多行字符串。

正确：

```
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(): ...
```

错误：

```
if (x):
    bar()
if not(x):
    bar()
return (foo)
```

3.4 缩进

使用 4 个空格缩进代码块。

永远不要使用制表符。隐式行延续应使换行的元素垂直对齐（参见行长度示例），或使用 4 个空格的悬挂缩进。关闭的（圆形、方形或花括号）括号可以放在表达式的末尾，或者放在单独的行上，但如果是单独的行，则应与对应的开括号所在行缩进相同。

正确：

```
# 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,
    ...
}
```

错误：

```
# 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,
    ...
}
```

3.4.1 序列项中的尾随逗号?

仅当序列中的结束容器标记]、) 或 } 未与最后一个元素出现在同一行时，或者对于只有一个元素的元组时，才推荐使用尾随逗号。尾随逗号的存在也被用作提示，指导我们的 Python 代码自动格式化工具 Black 或 Pyink 在最后一个元素后存在逗号时，将容器中的项自动格式化为每行一项。

正确:

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

错误:

```
golomb4 = [
    0,
    1,
    4,
    6,]
```

3.5 空行

顶级定义（无论是函数还是类定义）之间使用两个空行。方法定义之间以及类的文档字符串与第一个方法之间使用一个空行。`def` 行后不加空行。在函数或方法内部，根据你的判断适当地使用单个空行。

空行不需要固定在定义处。例如，在函数、类或方法定义之前的相关注释可以是有意义的。考虑你的注释是否作为文档字符串的一部分会更有用。

3.6 空格

遵循标点符号周围空格使用的标准排版规则。

括号、方括号或大括号内部不加空格。

正确:

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

错误:

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

在逗号、分号或冒号之前不加空格。在逗号、分号或冒号之后要加空格，除非是在行末。

正确:

```
if x == 4:
    print(x, y)
x, y = y, x
```

错误:

```
if x == 4 :
    print(x , y)
x , y = y , x
```

在开始参数列表、索引或切片的开括号/方括号之前不加空格。

正确:

```
spam(1)
```

错误：

```
spam (1)
```

正确：

```
dict['key'] = list[index]
```

错误：

```
dict ['key'] = list [index]
```

不要有尾随空格。

在赋值（=）、比较（==, <, >, !=, <>, <=, >=, in, not in, is, is not）和布尔运算（and, or, not）的二元运算符两侧各加一个空格。对于算术运算符（+, -, *, /, //, %, **, @）周围空格的插入，使用你的最佳判断。

正确：

```
x == 1
```

错误：

```
x<1
```

当 '=' 用于指示关键字参数或默认参数值时，如果存在类型注解，请不要在其周围使用空格。

正确：

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

错误：

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

不要使用空格来垂直对齐多行标记，因为它会成为维护负担（适用于 :, #, = 等）：

正确：

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

dictionary = {
```

```
    'foo': 1,
    'long_name': 2,
}
```

错误:

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

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

3.7 Shebang 行

大多数.py 文件不需要以 #! 行开头。程序的主文件应以 `#!/usr/bin/env python3`（以支持虚拟环境）或 `#!/usr/bin/python3` 开头，遵循 PEP-394[20]。

这一行由内核用来查找 Python 解释器，但在导入模块时会被 Python 忽略。只有在打算直接执行的文件上才需要这一行。

3.8 注释和文档字符串

确保您的代码有适当的注释和文档字符串。

3.8.1 文档字符串

Python 使用文档字符串（docstrings）来记录代码。文档字符串是包、模块、类或函数中的第一个语句。这些字符串可以通过对象的 `__doc__` 属性自动提取，并被 pydoc 使用。（可以尝试对你的模块运行 pydoc 来查看其效果。）始终使用三个双引号 """ 格式来编写文档字符串（根据 PEP 257[21]）。文档字符串应组织为一个摘要行（一行不超过 80 个字符），以句号、问号或感叹号结束。如果需要编写更多内容（鼓励这样做），摘要行后必须跟一个空行，然后是文档字符串的其余部分，从与首行第一个引号相同的游标位置开始。以下还有更多关于文档字符串的格式指南。

[20] https://peps.python.org/pep-0394/
[21] https://peps.python.org/pep-0257/

3.8.2 模块

每个文件都应包含许可证样板代码。选择项目使用的许可证对应的适当样板代码（例如，Apache 2.0、BSD、LGPL、GPL）。

文件应以描述模块内容和用法的文档字符串开头。

```
"""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.function_bar()
"""
```

3.8.2.1 测试模块 测试文件的模块级文档字符串不是必需的。仅当可以提供额外信息时才应包含文档字符串。

示例包括：有关如何运行测试的具体细节、非常规设置模式的解释、对外部环境的依赖性等。

```
"""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.
"""
```

不要使用没有提供任何新信息的文档字符串。

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

3.8.3 函数与方法

在本节中，“函数”指的是方法、函数、生成器或属性。

每个具有以下一个或多个特征的函数都必须包含文档字符串：

- 属于公共 API
- 非琐碎的规模
- 非显而易见的逻辑

文档字符串应提供足够的信息，以便在不阅读函数代码的情况下能够编写对该函数的调用。文档字符串应描述函数的调用语法及其语义，但通常不包括实现细节，除非这些细节与函数的使用方式相关。例如，如果函数作为副作用修改了其某个参数，应在文档字符串中注明。否则，与调用者无关的函数实现的微妙但重要的细节应作为代码旁边的注释，而不是在函数的文档字符串中表达。

文档字符串可以是描述性风格（`"""Fetches rows from a Bigtable."""`）或命令式风格（`"""Fetch rows from a Bigtable."""`），但在同一文件中风格应保持一致。`@property` 数据描述符的文档字符串应使用与属性或函数参数相同的风格（`"""The Bigtable path."""`，而不是 `"""Returns the Bigtable path."""`）。

函数的某些方面应在以下特定部分中记录。每个部分以一个标题行开始，标题行以冒号结尾。除了标题行外，所有部分应保持 2 或 4 个空格的悬挂缩进（在文件中保持一致）。如果函数的名称和签名足够清晰，可以通过一行文档字符串充分描述，则可以省略这些部分。

Args 按名称列出每个参数。参数名称后应跟一个描述，描述与名称之间用冒号加空格或换行符分隔。如果描述过长，无法在一行 80 个字符内适应，则使用比参数名称多 2 或 4 个空格的悬挂缩进（与文件中其他文档字符串保持一致）。如果代码中没有相应的类型注解，描述中应包含所需的类型信息。如果函数接受 `*foo`（可变长度参数列表）和/或 `**bar`（任意关键字参数），应将其列为 `*foo` 和 `**bar`。

Returns（或 Yields：对于生成器） 描述返回值（或生成器 yield 的值）的语义，包括类型注解未提供的任何类型信息。如果函数仅返回 None，则不需要此部分。如果文档字符串以 “Return”、“Returns”、“Yield” 或 “Yields” 开头（例如 `"""Returns row from Bigtable as a tuple of strings."""`），且开头的句子足以描述返回值，则也可以省略此部分。不要模仿较旧的 “NumPy 风格”（示例），这种风格常将元组返回值描述为多个单独命名的返回值（从未提及元组）。相反，应描述为：“Returns: A tuple (mat_a, mat_b), where mat_a is …, and …”。文档字符串中的辅助名称不必与函数体中使用的任何内部名称对应（因为这些不是 API 的一部分）。如果函数使用 yield（是生成器），Yields: 部分应记录 next() 返回的对象，而不是调用时评估的生成器对象本身。

Raises 列出与接口相关的所有异常，并附上描述。使用与 Args: 中描述的类似异常

名称 + 冒号 + 空格或换行符及悬挂缩进的风格。你不应记录因违反文档字符串中指定的 API 而引发的异常（因为这会矛盾地使违反 API 的行为成为 API 的一部分）。

```
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.
    """
```

类似地，`Args:` 部分的这种带换行符的变体也是允许的：

```
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.
    """
```

3.8.3.1 重写方法 如果一个方法明确使用 [@override][22] 装饰器（来自 typing_extensions 或 typing 模块）重写了基类中的方法，则不需要文档字符串，除非重写方法的行为实质上细化了基方法的契约，或者需要提供额外细节（例如，记录额外的副作用）。在这种情况下，重写方法的文档字符串至少需要描述这些差异。

[22] https://typing-extensions.readthedocs.io/en/latest/#override

```
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."""
```

3.8.4 类

类定义下方应有一个文档字符串，用于描述该类。公共属性（不包括属性装饰器，见属性）应在此处的 `Attributes` 部分中记录，并遵循与函数 `Args` 部分相同的格式。

```
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."""
```

所有类的文档字符串应以一行摘要开头，描述类实例代表的内容。这意味着异常的子类也应描述异常代表的内容，而不是其可能发生的上下文。类的文档字符串不应重复不必要的信息，例如说明该类是一个类。

正确：

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

  ...
  """

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

错误：

```
class CheeseShopAddress:
  """Class that describes the address of a cheese shop.

  ...
  """

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

3.8.5 块注释和行内注释

代码中复杂部分是添加注释的最后地方。如果你在下次 [代码审查][23]时需要解释某段代码，现在就应该为它添加注释。复杂的操作在开始前应有几行注释。非显而易见的代码应在行末添加注释。

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
```

[23] http://en.wikipedia.org/wiki/Code_review

```
# 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.
```

为了提高可读性，这些注释应至少与代码间隔 2 个空格，并以注释字符 # 开始，之后至少有一个空格，然后才是注释文本本身。

另一方面，永远不要描述代码本身。假设阅读代码的人比你更了解 Python（尽管他们不知道你试图做什么）。

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

3.8.6 标点、拼写和语法

注意标点、拼写和语法；编写良好的注释比编写糟糕的注释更容易阅读。

注释应像叙述性文本一样易于阅读，使用正确的首字母大写和标点。在许多情况下，完整的句子比句子片段更具可读性。较短的注释，例如代码行末尾的注释，有时可以不太正式，但你应保持风格的一致性。

尽管代码审查者指出你使用了逗号而应使用分号可能会让人感到沮丧，但源代码保持高度清晰度和可读性非常重要。正确的标点、拼写和语法有助于实现这一目标。

3.9 字符串

使用 f-string、% 运算符或 format 方法来格式化字符串，即使参数都是字符串。使用你的最佳判断来选择字符串格式化选项。使用 + 进行单次拼接是可以的，但不要使用 + 进行格式化。

```
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)
```

避免在循环中使用 + 和 += 运算符来累积字符串。在某些情况下，使用加法累积字符串可能导致二次方而不是线性运行时间复杂度。尽管在 CPython 上这种常见的累积操作可能会被优化，但这是一个实现细节。优化适用的条件不易预测且可能会发生变化。相反，应将每个子字符串添加到一个列表中，并在循环结束后使用 ''.join 方法连接列表，或者将每个子字符串写入 io.StringIO 缓冲区。这些技术始终具有摊销线性运行时间复杂度。

```
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>'
```

在一个文件中保持字符串引号字符的选择一致。选择单引号 ' 或双引号 " 并坚持使用。可以在字符串中使用另一种引号字符，以避免在字符串中对引号字符进行反斜杠转义。

```
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.")
```

优先使用 """ 来表示多行字符串，而不是 '''。项目可以选择对所有非文档字符串的多行字符串使用 '''，但前提是它们也对普通字符串使用 '。无论如何，文档字符串必须使用 """。

多行字符串不会跟随程序其余部分的缩进。如果需要避免在字符串中嵌入额外的空格，可以使用拼接的单行字符串，或者使用 [textwrap.dedent()][24] 处理多行字符串

[24] https://docs.python.org/3/library/textwrap.html#textwrap.dedent

以移除每行的初始空格：

```
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.""")
```

请注意，在此处使用反斜杠并不违反禁止显式行延续的规定；在这种情况下，反斜杠是在字符串字面量中转义换行符。

3.9.1 日志记录

对于期望将模式字符串（包含%-占位符）作为第一个参数的日志函数：始终使用字符串字面量（而非 f-string!）作为第一个参数，并将模式参数作为后续参数。某些日志实现会收集未展开的模式字符串作为一个可查询字段。这还可以避免花费时间渲染一条没有配置输出目标的日志消息。

```
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
        }')
```

3.9.2 错误消息

错误消息（例如：ValueError 等异常中的消息字符串，或显示给用户的消息）应遵循以下三条准则：

1. 消息需要精确匹配实际的错误条件。
2. 插值部分需要始终清晰可辨。
3. 应便于简单的自动化处理（例如通过 grep 查找）。

```
  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 文件、套接字和类似的有状态资源

在使用完文件和套接字后，显式地关闭它们。这一规则自然扩展到内部使用套接字的可关闭资源，例如数据库连接，以及其他需要类似方式关闭的资源。例如，这包括 mmap 映射[25]、h5py 文件对象[26]和 matplotlib.pyplot 图形窗口[27]等。

不必要地保持文件、套接字或其他此类有状态对象的开启状态有许多缺点：

- 它们可能消耗有限的系统资源，例如文件描述符。处理大量此类对象的代码如果在使用后不及时将资源归还系统，可能会不必要地耗尽这些资源。
- 保持文件开启可能会阻止其他操作，例如移动或删除文件，或者卸载文件系统。
- 在整个程序中共享的文件和套接字可能在逻辑上关闭后被意外读取或写入。如果它们被实际关闭，尝试读取或写入将引发异常，从而更快地暴露问题。

此外，尽管文件、套接字（以及一些行为类似的其他资源）在对象销毁时会自动关闭，但将对象的生命周期与资源的状态绑定是一种不好的做法：

- 无法保证运行时会在何时实际调用 `__del__` 方法。不同的 Python 实现使用不同的内存管理技术，例如延迟垃圾回收，可能会任意且无限期地延长对象的生命周期。

[25] https://docs.python.org/3/library/mmap.html
[26] https://docs.h5py.org/en/stable/high/file.html
[27] https://matplotlib.org/2.1.0/api/_as_gen/matplotlib.pyplot.close.html

- 对文件的意外引用（例如在全局变量或异常跟踪中）可能使其存活时间超出预期。
- 依赖终结器（finalizers）进行具有可观察副作用的自动清理已被多次证明会导致重大问题，这种问题跨越了数十年和多种语言（参见关于 Java 的文章[28]）。

管理文件和类似资源的首选方式是使用 `with` 语句[29]：

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

对于不支持 `with` 语句的类文件对象，使用 `contextlib.closing()`：

```
import contextlib

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

在极少数情况下，如果基于上下文的资源管理不可行，代码文档必须清楚地解释资源生命周期是如何管理的。

3.11 TODO 注释

对临时代码、短期解决方案或足够好但不完美的代码使用 `TODO` 注释。

`TODO` 注释以全大写的 `TODO` 开头，后跟冒号，以及一个指向包含上下文的资源的链接，最好是一个错误（bug）引用。错误引用更可取，因为错误会被跟踪并有后续评论。在上下文之后，紧跟一个以连字符 - 开头的解释性字符串。目的是使用一致的 `TODO` 格式，以便搜索以获取更多细节。

示例：

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

旧风格（以前推荐但不鼓励在新代码中使用）：

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

[28] https://wiki.sei.cmu.edu/confluence/display/java/MET12-J.+Do+not+use+finalizers

[29] http://docs.python.org/reference/compound_stmts.html#the-with-statement

避免添加以个人或团队作为上下文的 TODO：

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

如果你的 TODO 是“在未来某个日期做某事”的形式，确保包含非常具体的日期（例如“2009 年 11 月前修复”）或非常具体的事件（例如“当所有客户端都能处理 XML 响应时移除此代码”），以便未来的代码维护者能够理解。使用问题（issues）来跟踪这些情况是理想的。

3.12 导入格式

导入语句应独占一行，但 typing 和 collections.abc 的导入可以例外（见用于类型的导入）。

示例：

正确：

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

错误：

```
import os, sys
```

导入语句始终放在文件顶部，仅在模块注释和文档字符串之后，模块全局变量和常量之前。导入应按从最通用到最不通用的顺序分组：

1. Python 未来导入语句。例如：

   ```
   from __future__ import annotations
   ```

   有关详情见现代 Python: from __future__ import。

2. Python 标准库导入。例如：

   ```
   import sys
   ```

3. 第三方模块或包导入。例如：

   ```
   import tensorflow as tf
   ```

4. 代码仓库子包导入。例如：

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

5. 已弃用：与当前文件属于同一顶级子包的应用程序特定导入。例如：

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

   你可能会在较旧的 Google Python 风格代码中看到这种做法，但这不再是必需的。鼓励新代码不再遵循此方式，简单地将应用程序特定的子包导入视为与其他子包导入相同。

在每个分组内，导入应按模块的完整包路径（`from path import ...` 中的路径）进行字典序排序，忽略大小写。代码可以在导入分组之间选择性地添加空行。

```
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 语句

通常每行只有一个语句。

然而，只有当整个语句适合放在一行时，你才可以将测试结果与测试放在同一行。特别是，`try/except` 语句永远不能这样做，因为 `try` 和 `except` 无法同时放在一行；对于 `if` 语句，只有在没有 `else` 的情况下才可以这样做。

正确：

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

错误：

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

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

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

3.14 Getters 和 Setters

应在获取或设置变量值时具有重要作用或行为时使用 getter 和 setter 函数（也称为访问器和修改器，accesors and mutators）。

特别是，当获取或设置变量的操作复杂或成本显著时（无论是当前还是在合理的未来），应使用 getter 和 setter 函数。

例如，如果一对 getter/setter 仅简单地读取和写入内部属性，则应将该内部属性设为公共属性。相比之下，如果设置变量意味着某些状态失效或需要重建，则应使用 setter 函数。函数调用暗示可能正在进行非平凡操作。或者，当需要简单逻辑时，可以考虑使用属性（properties，见属性），或重构代码以不再需要 getter 和 setter。

Getter 和 setter 函数应遵循命名准则，例如 `get_foo()` 和 `set_foo()`。

如果过去的做法允许通过属性访问，不要将新的 getter/setter 函数绑定到该属性。任何仍尝试通过旧方法访问变量的代码应明显报错，以便开发者意识到复杂性的变化。

3.15 命名

命名应具有描述性。这包括函数、类、变量、属性、文件以及任何其他类型的命名实体。

避免使用缩写。特别是，不要使用对项目外部读者来说模糊或不熟悉的缩写，也不要通过删除单词中的字母来缩写。

始终使用 `.py` 文件扩展名。不要使用破折号。

命名示例:

- 模块名: `module_name`
- 包名: `package_name`
- 类名: `ClassName`
- 方法名: `method_name`
- 异常名: `ExceptionName`
- 函数名: `function_name`
- 全局常量名: `GLOBAL_CONSTANT_NAME`
- 全局变量名: `global_var_name`
- 实例变量名: `instance_var_name`
- 函数参数名: `function_parameter_name`
- 局部变量名: `local_var_name`
- 查询专有名词: `query_proper_noun_for_thing`
- 通过 HTTPS 发送缩写: `send_acronym_via_https`

3.15.1 要避免的名称

- 除非在特定允许的情况下，避免使用单字符命名，允许的单字符命名情况:
  - 计数器或迭代器（例如: `i`, `j`, `k`, `v` 等）
  - `try/except` 语句中的异常标识符 `e`
  - `with` 语句中的文件句柄 `f`
  - 无约束的私有类型变量（例如: `_T = TypeVar("_T")`, `_P = ParamSpec("_P")`）
  - 与参考论文或算法中已建立的数学符号相匹配的名称（参见数学符号）

  请注意不要滥用单字符命名。一般来说，命名的描述性应与名称的可见性范围成正比。例如，`i` 对于一个 5 行的代码块可能是合适的，但在多个嵌套作用域中，它可能过于模糊。
- 任何包/模块名中使用破折号（`-`）
- `__double_leading_and_trailing_underscore__` 的名称（Python 保留）
- 冒犯性术语
- 不必要地包含变量类型的名称（例如: `id_to_name_dict`）

3.15.2 命名约定

- “内部” 指的是模块内部，或者类中的受保护或私有成员。
- 在模块变量或函数名前添加单个下划线（_）在一定程度上可以保护这些成员（代码检查工具会标记对受保护成员的访问）。注意，单元测试访问被测试模块中的受保护常量是可以的。
- 在实例变量或方法名前添加双下划线（__，即 “dunder”）会通过名称改编（name mangling）使其对类有效私有；我们不鼓励使用这种方式，因为它会影响可读性和可测试性，而且实际上并非真正私有。优先使用单个下划线。
- 将相关的类和顶级函数放在同一个模块中。与 Java 不同，Python 不需要限制每个模块只包含一个类。
- 类名使用 `CapWords` 风格，模块名使用 `lower_with_under.py` 风格。尽管有一些旧模块使用 `CapWords.py` 命名，但现在不鼓励这样做，因为当模块名与类名相同时会引起混淆。（例如：“等等——我写的是 `import StringIO` 还是 `from StringIO import StringIO`？”）
- 新的单元测试文件应遵循 PEP 8 兼容的 `lower_with_under` 方法命名，例如 `test_<method_under_test>_<state>`。为了与遵循 `CapWords` 函数命名的旧模块保持一致性（*），方法名中可以包含下划线以分隔名称的逻辑组件。一个可能的模式是 `test<MethodUnderTest>_<state>`。

3.15.3 文件命名

Python 文件名必须使用 `.py` 扩展名，且不得包含破折号（-）。这确保它们可以被导入和进行单元测试。如果希望可执行文件无需扩展名即可访问，请使用符号链接或一个包含 `exec "$0.py" "$@"` 的简单 bash 包装脚本。

3.15.4 来自 Guido 推荐的指导原则

表 1: Guido’s Recommendations

Type	Public	Internal
Packages	`lower_with_under`	
Modules	`lower_with_under`	`_lower_with_under`
Classes	`CapWords`	`_CapWords`

Type	Public	Internal
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`	

3.15.5 数学符号

对于数学计算密集的代码，如果变量名符合参考论文或算法中的既定符号，优先使用短变量名，即使这些名称可能违反风格指南。

在使用基于既定符号的名称时：

- 在注释或文档字符串中引用所有命名约定的来源，最好包含指向学术资源本身的超链接。如果来源不可访问，需清晰记录命名约定。
- 对于公共 API，优先使用符合 PEP 8 的描述性名称（`descriptive_names`），因为这些名称更可能在无上下文的情况下被遇到。
- 使用范围狭窄的 `pylint: disable=invalid-name` 指令来抑制警告。对于仅几个变量的情况，在每个变量的行末使用该指令作为注释；对于更多变量，在代码块开头应用该指令。

3.16 main 函数

在 Python 中，`pydoc` 和单元测试要求模块是可导入的。如果一个文件旨在作为可执行文件使用，其主要功能应放在 `main()` 函数中，并且你的代码应始终在执行主程序前检查 `if __name__ == '__main__'`，以确保模块被导入时不会执行主程序。

当使用 `absl` 时，使用 `app.run`：

```
from absl import app
...
```

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

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

否则，使用以下结构：

```python
def main():
    ...

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

模块顶级的代码在模块被导入时会被执行。注意不要在文件被 pydoc 处理时调用函数、创建对象或执行其他不应执行的操作。

3.17 函数长度

偏好小型且专注的函数。

我们认识到有时候长函数也是合适的，因此没有对函数长度设置硬性限制。如果一个函数超过大约 40 行，请考虑是否可以在不破坏程序结构的情况下将其拆分。

即使你的长函数现在运行得很好，几个月后有人修改它时可能会添加新行为。这可能导致难以发现的错误。保持函数短小简单可以让其他人更容易阅读和修改你的代码。

在处理一些代码时，你可能会遇到长而复杂的函数。不要害怕修改现有代码：如果发现处理这样的函数很困难、错误难以调试，或者你想在多个不同上下文中使用其中的一部分，考虑将函数拆分成更小、更易管理的部分。

3.18 类型注解

3.18.1 一般规则

- 熟悉类型提示[30]。

[30] https://docs.python.org/3/library/typing.html

- 通常不需要对 `self` 或 `cls` 进行类型注解。如果为了提供正确的类型信息而有必要，可以使用 `Self`[31]，例如：

```
from typing import Self

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

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

- 同样，不要觉得必须为 `__init__` 的返回值添加类型注解（因为 `None` 是唯一有效的选项）。
- 如果某个变量或返回类型不宜明确表达，请使用 `Any`。
- 你无需为模块中的所有函数都添加类型注解。
  - 至少为你的公共 API 添加类型注解。
  - 在安全性和清晰度与灵活性之间找到一个良好的平衡，需运用判断力。
  - 为容易出现类型相关错误的代码（例如之前的 bug 或复杂代码）添加类型注解。
  - 为难以理解的代码添加类型注解。
  - 为从类型角度来看已经稳定的代码添加类型注解。在许多情况下，你可以为成熟代码中的所有函数添加类型注解，而不会损失太多灵活性。

3.18.2 换行

遵循现有的缩进规则，见缩进。

在添加类型注解后，许多函数签名会变成“每行一个参数”。为了确保返回类型也独占一行，可以在最后一个参数后添加逗号。

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

[31] https://docs.python.org/3/library/typing.html#typing.Self

总是优先在变量之间换行，而不是例如在变量名和类型注解之间换行。然而，如果所有内容都能适应同一行，就直接写在一行。

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

如果函数名、最后一个参数和返回类型组合起来太长，则在新行中缩进 4 个空格。使用换行时，优先将每个参数和返回类型放在单独的行，并将右括号与 `def` 对齐：

正确：

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

可选地，返回类型可以与最后一个参数放在同一行：

可以：

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

pylint 允许将右括号移到新行并与左括号对齐，但这种方式可读性较差。

错误：

```
def my_method(self,
              other_arg: MyLongType | None,
             ) -> dict[OtherLongType, MyLongType]:
    ...
```

如上例所示，尽量不要拆分类型。然而，有时类型太长，无法放在一行（尽量保持子类型不被拆分）。

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

如果单个变量名和类型组合太长，考虑为类型使用别名。最后的解决办法是在冒号后换行并缩进 4 个空格。

正确：

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

错误：

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

3.18.3 前向声明

如果需要使用尚未定义的类名（来自同一模块），例如在类声明中需要使用该类名，或者使用在代码中稍后定义的类，可以选择以下两种方式之一：使用 `from __future__ import annotations` 或将类名作为字符串。

正确：

```
from __future__ import annotations

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

class OtherClass:
    ...
```

正确：

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

class OtherClass:
    ...
```

3.18.4 默认值

根据 PEP 008[32]，只有在参数同时具有类型注解和默认值时，才在 = 周围使用空格。

正确：

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

错误：

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

3.18.5 NoneType

在 Python 类型系统中，NoneType 是一个 “一级” 类型，在类型注解中，None 是 NoneType 的别名。如果一个参数可以是 None，必须明确声明！你可以使用 | 联合类型表达式（在 Python 3.10+ 的新代码中推荐），或者使用较旧的 Optional 和 Union 语法。

应使用显式的 X | None 而非隐式的。早期的类型检查器允许将 a: str = None 解释为 a: str | None = None，但这不再是推荐的行为。

正确：

```
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:
    ...
```

错误：

```
def nullable_union(a: Union[None, str]) -> str:
    ...
def implicit_optional(a: str = None) -> str:
    ...
```

[32] https://peps.python.org/pep-0008/#other-recommendations

3.18.6 类型别名

你可以声明复杂类型的别名。别名的名称应采用 `CapWorded` 形式。如果别名仅在本模块中使用，它应该是 `_Private`。

注意，`: TypeAlias` 注解仅在 Python 3.10+ 版本中支持。

```
from typing import TypeAlias

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

3.18.7 忽略类型

你可以使用特殊注释 `# type: ignore` 来禁用某一行上的类型检查。

`pytype` 提供了一个针对特定错误的禁用选项（类似于 lint）：

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

3.18.8 类型化变量

带注解的赋值 如果一个内部变量的类型难以或无法推断，请使用带注解的赋值来指定其类型——在变量名和值之间使用冒号和类型（与为具有默认值的函数参数指定类型的方式相同）：

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

类型注释 虽然在代码库中可能仍会看到类型注释（在 Python 3.6 之前是必需的），但不要再添加行末的 `# type: <type name>` 注释：

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

3.18.9 元组 vs 列表

类型化的列表只能包含单一类型的对象。类型化的元组可以具有单一重复类型，或者具有固定数量的、类型各异的元素。后者通常用作函数的返回类型。

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

3.18.10 类型变量

Python 类型系统支持泛型，见泛型。类型变量（如 TypeVar 和 ParamSpec）是使用泛型的常用方式。

示例：

```
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
```

类型变量的约束：

TypeVar 可以被约束：

```
AddableType = TypeVar("AddableType", int, float, str)

def add(a: AddableType, b: AddableType) -> AddableType:
    return a + b
```

3.19 预定义类型变量 AnyStr：

typing 模块中常用的预定义类型变量是 AnyStr，用于可以是 bytes 或 str 且必须是同一类型的多个注解。

```
from typing import AnyStr

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

类型变量命名规则：

类型变量必须具有描述性名称，除非它满足以下所有条件：- 不在外部可见 - 未被约束

正确：

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

错误：

```
T = TypeVar("T")
P = ParamSpec("P")
_T = TypeVar("_T", int, float, str)
_F = TypeVar("_F", bound=Callable)
```

3.19.1 字符串类型

在新代码中不要使用 `typing.Text`，它仅用于 Python 2/3 兼容性。

对于字符串/文本数据，使用 `str`。对于处理二进制数据的代码，使用 `bytes`。

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

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

如果函数的所有字符串类型始终相同，例如上述代码中返回类型与参数类型相同，使用 `AnyStr`。

```
from typing import AnyStr

def process_string(x: AnyStr) -> AnyStr:
    ...
```

3.19.2 用于类型的导入

对于用于支持静态分析和类型检查的 `typing` 或 `collections.abc` 模块中的符号（包括类型、函数和常量），始终直接导入符号本身。这样可以使常用注解更简洁，并与全球通用的类型注解实践保持一致。明确允许从 `typing` 和 `collections.abc` 模块在一行中导入多个特定符号。例如：

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

由于这种导入方式会将符号添加到本地命名空间，`typing` 或 `collections.abc` 中的名称应被视为类似于关键字的地位，不应在 Python 代码中（无论是否带类型注解）定义这些名称。如果模块中存在类型与现有名称的冲突，使用 `import x as y` 方式导入。

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

在为函数签名添加类型注解时，优先使用抽象容器类型（如 `collections.abc.Sequence`），而不是具体类型（如 `list`）。如果需要使用具体类型（例如，带类型元素的元组），优先使用内置类型（如 `tuple`），而不是 `typing` 模块中的参数化类型别名（例如，`typing.Tuple`）。

```
from typing import List, Tuple

def transform_coordinates(original: List[Tuple[float, float]]) ->
    List[Tuple[float, float]]:
     ...
```

```
from collections.abc import Sequence

def transform_coordinates(original: Sequence[tuple[float, float]])
    -> Sequence[tuple[float, float]]:
     ...
```

3.19.3 条件导入

仅在特殊情况下使用条件导入，即当需要避免在运行时加载仅用于类型检查的额外导入时。这种模式不被鼓励，应优先考虑替代方案，例如重构代码以允许顶层导入。

仅用于类型注解的导入可以放在 `if TYPE_CHECKING:` 块中。

条件导入的类型需要以字符串形式引用，以确保与 Python 3.6 的向前兼容，因为在 Python 3.6 中注解表达式会被实际执行。只有专门用于类型注解的实体（包括别名）才应在此定义；否则会导致运行时错误，因为该模块在运行时不会被导入。该块应紧跟在所有常规导入之后。类型导入列表中不应有空行。该列表的排序应如同常规导入列表一样。

```
import typing
```

```
if typing.TYPE_CHECKING:
    import sketch

def f(x: "sketch.Sketch") -> None:
    ...
```

3.19.4 循环依赖

由于类型注解导致的循环依赖是一种代码异味（code smell）。此类代码是重构的理想候选。虽然技术上可以保留循环依赖，但许多构建系统不允许这样做，因为每个模块都必须依赖另一个模块。

将导致循环依赖导入的模块替换为 `Any`。使用一个有意义的别名，并使用该模块中的真实类型名称（`Any` 的任何属性都是 `Any`）。别名定义应与最后一个导入之间隔一行。

```
from typing import Any

some_mod = Any  # some_mod.py imports this module.

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

3.19.5 泛型

在进行类型注解时，优先为泛型类型的参数列表指定类型参数；否则，泛型的参数将被假定为 `Any`。

正确：

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

错误：

```
# 这被解释为 get_names(employee_ids: Sequence[Any]) -> Mapping[Any,
    Any]
def get_names(employee_ids: Sequence) -> Mapping:
    ...
```

如果泛型的最佳类型参数是 `Any`，请明确指定，但请记住，在许多情况下 `TypeVar` 可能更合适：

错误：

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

正确：

```
_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 结束语

保持一致。

如果您正在编辑代码，请花几分钟查看周围的代码并确定其风格。如果它们在索引变量名称中使用 `_idx` 后缀，您也应该这样做。如果它们的注释周围有小盒子的哈希标记，请使您的注释也有小盒子的哈希标记。

拥有风格指南的目的是有一个共同的编码词汇，以便人们可以专注于您在说什么而不是您如何说。我们在这里呈现全球风格规则，以便人们知道词汇，但本地风格也很重要。如果您添加到文件中的代码看起来与周围的现有代码大不相同，当他们去阅读它时会打断读者的节奏。

但是，一致性有界限。它在本地更重，并且在全球风格未指定的选择上。一致性不应一般用作在不考虑新风格益处或代码库趋向于更新风格的情况下以旧风格做事的理由。