Python 代码风格指南

Guido van Rossum, Barry Warsaw, Alyssa Coghlan

2025 年 04 月 04 日

目录

1 引言

本文档为 Python 主发行版标准库中的 Python 代码提供了编码规范。请参阅配套的信息性 PEP，描述了 Python C 实现中 C 代码的风格指南[1]。

本文档和 PEP 257[1]（文档字符串规范）改编自 Guido 最初的 Python 风格指南文章，并从 Barry 的风格指南[2]中加入一些补充内容。

随着时间的推移，随着新的规范被识别以及语言本身的变更使旧的规范变得过时，本风格指南会不断演变。

许多项目有自己的编码风格指南。如果发生冲突，项目特定的指南在该项目中优先。

2 固守愚蠢的连贯实为狭隘心智之魔障 [2]

Guido 的一个关键见解是，代码被阅读的频率远高于被编写的频率。本文提供的指南旨在提高代码的可读性，并使 Python 代码在广泛的范围内保持一致。正如 PEP 20[3] 所说，“可读性很重要”。

风格指南的核心是一致性。与本风格指南保持一致很重要。在一个项目内保持一致更重要。在一个模块或函数内保持一致最为重要。

然而，要知道何时可以不一致——有时候风格指南的建议并不适用。如有疑问，使用你的最佳判断，参考其他示例并决定哪种方式看起来最好。不要犹豫，随时提问！

特别注意：不要仅仅为了遵循本 PEP 而破坏向后兼容性！

以下是忽略某些指南的合理理由：

1. 遵循指南会使代码可读性降低，即使是对习惯阅读遵循本 PEP 的代码的人。

[1] https://peps.python.org/pep-0257/

[2] 出自思想家爱默生（Ralph Waldo Emerson）《论自助（Self-Reliance）》名句，意思是，盲目地坚持自己的想法，而不愿意根据新的知识或情况来改变，是一种狭隘、缺乏远见的表现。该译本由已故翻译家屠岸先生审定，被收录于《爱默生选集》（人民文学出版社 1993 年版），是哲学翻译领域的典范之作。（译者注）

[3] https://peps.python.org/pep-0020/

2. 为与周围违反指南的代码（可能是历史原因）保持一致——尽管这也是清理他人遗留问题（以真正的 XP 风格）的机会。

3. 代码早于指南引入，且没有其他理由修改该代码。

4. 代码需要与不支持风格指南推荐功能的旧版 Python 保持兼容。

3 代码布局

3.1 缩进

每级缩进使用 4 个空格。

续行应通过 Python 在括号、方括号和大括号内的隐式行连接来垂直对齐包裹元素，或者使用 悬挂缩进 [4]。使用悬挂缩进时，应考虑以下事项：第一行不应包含参数，且应使用额外的缩进来清楚地区分续行：

```
# 正确：

# 与开始分隔符对齐。
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 添加 4 个空格（额外的缩进级别）以区分参数和其他内容。
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 悬挂缩进应增加一级缩进。
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```
# 错误：

# 不使用垂直对齐时，禁止在第一行放置参数。
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

[4] 悬挂缩进是一种排版风格，其中段落的所有行都缩进，除了第一行。在 Python 上下文中，该术语用于描述一种风格，其中括号语句的左括号是行的最后一个非空白字符，后续行缩进直到右括号。

```
# 需要进一步缩进，因为缩进无法区分。
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

续行的 4 空格规则是可选的。

可选：

```
# 正确：

# 悬挂缩进 *可以* 不使用 4 个空格。
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

当if 语句的条件部分长到需要跨多行编写时，值得注意的是，两个字符的关键字（即 if），加上一个空格，再加上一个左括号，会为后续多行条件语句自然形成 4 个空格的缩进。这可能会与 if 语句内嵌套的代码块（也自然缩进 4 个空格）产生视觉冲突。本 PEP 对如何（或是否）进一步视觉区分条件行与 if 语句内的嵌套代码块未明确规定。以下是可接受的选项，包括但不限于：

```
# 正确：

# 无额外缩进。
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# 添加注释，在支持语法高亮的编辑器中提供一些区分。
if (this_is_one_thing and
    that_is_another_thing):
    # 由于两个条件都为真，我们可以执行 frobnicate。
    do_something()

# 在条件续行上添加额外缩进。
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

（另见下文关于在二元运算符前后换行的讨论。）

多行结构的结束括号/方括号/圆括号可以与列表最后一行第一个非空白字符对齐，如下：

```
# 正确：
```

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

或者与开始多行结构的行的第一个字符对齐，如下：

```
# 正确：

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

3.2 制表符还是空格?

空格是首选的缩进方法。

制表符仅用于与已使用制表符缩进的代码保持一致。

Python 不允许混用制表符和空格进行缩进。

3.3 最大行长度

所有行限制在最多 79 个字符。

对于结构限制较少的流动文本块（文档字符串或注释），行长度应限制在 72 个字符。

限制编辑器窗口宽度使得可以并排打开多个文件，并且在使用并列显示两个版本的代码审查工具时效果良好。

大多数工具的默认换行会破坏代码的视觉结构，使其更难理解。选择这些限制是为了避免在窗口宽度设置为 80 的编辑器中换行，即使工具在换行时在最后一列放置标记字符。一些基于 Web 的工具可能完全不提供动态换行。

一些团队强烈偏好更长的行长度。对于由团队主要或专门维护的代码，如果团队就此问题达成一致，可以将行长度限制增加到 99 个字符，但注释和文档字符串仍需在 72 个字符处换行。

Python 标准库较为保守，要求将行限制在 79 个字符（文档字符串/注释为 72 个字符）。

换行长行的首选方式是利用 Python 在括号、方括号和大括号内的隐式行连接。长行可以通过在括号内包装表达式来分成多行。应优先使用这些方式，而不是使用反斜杠进行行继续。

反斜杠在某些情况下仍然适用。例如，在 Python 3.10 之前，长的多重 `with` 语句无法使用隐式续行，因此反斜杠在这种情况下是可接受的：

```
# 可能：

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

（请参阅上文关于多行条件语句的讨论，了解更多关于多行 `with` 语句缩进的思考。）

另一个这样的情况是 `assert` 语句。

确保适当地缩进续行。

3.4 二元运算符前还是后换行？

几十年来，推荐的风格是在二元运算符后换行。但这在两方面损害可读性：运算符分散在屏幕的不同列，每运算符与其操作数分开，移到前一行。这迫使眼睛做额外的工作来分辨哪些项是加的，哪些是减的：

```
# 错误：
# 运算符与其操作数距离较远
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

为解决此可读性问题，数学家及其出版商遵循相反的惯例。Donald Knuth 在其 *Computers and Typesetting* 系列中解释了传统规则：“尽管段落内的公式总是在二元运算和关系后换行，展示的公式总是在二元运算前换行”[3]。

遵循数学传统通常会产生更易读的代码：

```
# 正确：
# 易于匹配运算符与操作数
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

在 Python 代码中，允许在二元运算符前后换行，只要局部保持一致。对于新代码，建议采用 Knuth 的风格。

3.5 空行

顶级函数和类定义前后用两个空行分隔。

类内的方法定义用单个空行分隔。

可以（谨慎地）使用额外的空行来分隔相关函数组。在一组相关的单行代码（例如一组虚拟实现）之间可以省略空行。

在函数中，谨慎使用空行来表示逻辑段落。

Python 接受 Control-L（即 ^L）换页符作为空白字符；许多工具将这些字符视为页面分隔符，因此你可以用它们来分隔文件的相关部分页面。注意，一些编辑器和基于 Web 的代码查看器可能无法识别 Control-L 作为换页符，而是显示其他符号。

3.6 源文件编码

Python 主发行版中的代码应始终使用 UTF-8 编码，且不应有编码声明。

在标准库中，非 UTF-8 编码仅用于测试目的。谨慎使用非 ASCII 字符，优先用于表示地点和人名。如果使用非 ASCII 字符作为数据，避免使用干扰性 Unicode 字符（如 z̨algo 和字节顺序标记）。

Python 标准库中的所有标识符必须使用仅限 ASCII 的标识符，并且在可行的情况下应使用英语单词（在许多情况下会使用非英语的缩写和技术术语）。

面向全球受众的开源项目鼓励采用类似策略。

3.7 导入

- 导入通常应放在单独的行：

```
# 正确：
import os
import sys
```

```
# 错误：
import sys, os
```

不过这样说也可以：

```
# 正确：
from subprocess import Popen, PIPE
```

- 导入始终放在文件顶部，紧跟模块注释和文档字符串之后，模块全局变量和常量之前。

  导入应按以下顺序分组：

  1. 标准库导入。
  2. 相关第三方导入。
  3. 本地应用程序/库特定的导入。

  每组导入之间应有一个空行。

- 推荐使用绝对导入，因为它们通常更易读，并且在导入系统配置错误（例如包内的目录出现在 `sys.path` 中）时行为更好（或至少提供更好的错误信息）：

```
# 正确：
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

然而，显式相对导入是绝对导入的可接受替代方案，特别是在处理复杂包布局时，使用绝对导入会显得过于冗长：

```
# 正确：
from . import sibling
from .sibling import example
```

标准库代码应避免复杂的包布局，始终使用绝对导入。

- 从包含类的模块导入类时，通常可以这样写：

```
# 正确：
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

如果这种写法导致本地名称冲突，则显式拼写：

```
# 正确：
import myclass
import foo.bar.yourclass
```

并使用 `myclass.MyClass` 和 `foo.bar.yourclass.YourClass`。

- 应避免使用通配符导入（`from <module> import *`），因为它们会使命名空间中的名称不清楚，混淆读者和许多自动化工具。通配符导入的一个可辩护用例是将内部接口重新发布为公共 API 的一部分（例如，用可选加速器模块的定义覆盖纯 Python 实现的接口，且事先不知道哪些定义会被覆盖）。

  以这种方式重新发布名称时，仍应遵循下文关于公共和内部接口的指南。

3.8 模块级双下划线名称

模块级的“双下划线”（即以两个前导和两个尾随下划线命名的名称），如 `__all__`、`__author__`、`__version__` 等，应放在模块文档字符串之后，但在任何导入语句之前，除了 `from __future__` 导入。Python 要求 future 导入必须出现在模块中除文档字符串外的任何其他代码之前：

```
# 正确：

"""这是一个示例模块。

此模块执行某些操作。
"""

from __future__ import barry_as_FLUFL

__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'

import os
import sys
```

4 字符串引号

在 Python 中，单引号字符串和双引号字符串是相同的。本 PEP 对此不做推荐。选择一种规则并坚持使用。然而，当字符串包含单引号或双引号字符时，使用另一种引号以避免在字符串中使用反斜杠。这会提高可读性。

对于三重引号字符串，始终使用双引号字符，以与 PEP 257 中的文档字符串规范保持一致。

5 表达式和语句中的空白

5.1 特别反感的小习惯

在以下情况下避免多余的空白：

- 括号、方括号或大括号内部：

```
# 正确：
spam(ham[1], {eggs: 2})
```

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

- 尾随逗号与随后关闭括号之间：

```
# 正确：
foo = (0,)
```

```
# 错误：
bar = (0, )
```

- 逗号、分号或冒号之前：

```
# 正确：
if x == 4: print(x, y); x, y = y, x
```

```
# 错误：
if x == 4 : print(x , y) ; x , y = y , x
```

- 然而，在切片中，冒号的作用类似于二元运算符，应在其两侧有相等的空白（将其视为优先级最低的运算符）。在扩展切片中，两个冒号必须应用相同数量的空

白。例外：当省略切片参数时，空白也被省略：

```
# 正确：
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]
```

```
# 错误：
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : step]
ham[ : upper]
```

- 在启动函数调用参数列表的左括号之前：

```
# 正确：
spam(1)
```

```
# 错误：
spam (1)
```

- 在启动索引或切片的左括号之前：

```
# 正确：
dct['key'] = lst[index]
```

```
# 错误：
dct ['key'] = lst [index]
```

- 在赋值（或其他）运算符周围使用多个空格以与其他行对齐：

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

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

5.2 其他建议

- 避免在任何地方使用尾随空白。因为它通常不可见，可能会造成混淆：例如，反斜杠后跟空格和换行符不算行继续标记。一些编辑器不会保留尾随空白，许多项目（如 CPython 本身）有预提交钩子来拒绝它。
- 始终在以下二元运算符两侧使用单个空格：赋值（=）、增量赋值（+=、-= 等）、比较（==、<、>、!=、<=、>=、in、not in、is、is not）、布尔运算（and、or、not）。
- 如果使用优先级不同的运算符，考虑在优先级最低的运算符周围添加空白。使用你自己的判断；但永远不要使用超过一个空格，并始终在二元运算符两侧使用相同数量的空白：

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

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

- 函数注解应使用冒号的常规规则,并在存在 -> 箭头时始终在其周围使用空格。(详见下文的 函数注解 部分)：

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

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

- 当用于指示关键字参数或未注解函数参数默认值时，不要在 = 号周围使用空格：

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

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

然而，当将参数注解与默认值结合使用时，在 = 号周围使用空格：

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

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

- 复合语句（同一行上的多条语句）通常不鼓励使用：

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

不推荐：

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

- 有时将带有小主体的 `if/for/while` 放在同一行是可以的，但永远不要对多子句语句这样做。也要避免折叠这样的长行！

不推荐：

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

绝对不要：

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

try: something()
finally: cleanup()
```

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

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

6 何时使用尾随逗号

尾随逗号通常是可选的，但在一个元素的元组中是强制性的。为清晰起见，建议将后者用（技术上冗余的）括号括起来：

```
# 正确：
FILES = ('setup.cfg',)
```

```
# 错误：
FILES = 'setup.cfg',
```

当尾随逗号是冗余的时，它们在版本控制系统使用时通常很有帮助，当值、参数或导入项列表预计会随时间扩展时。模式是将每个值（等）放在单独一行，始终添加尾随逗号，并在下一行添加关闭括号/方括号/大括号。然而，在与关闭分隔符同一行上使用尾随逗号没有意义（除了上述单元素元组的情况）：

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

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

7 注释

与代码矛盾的注释比没有注释更糟。始终优先保持注释与代码更改同步！

注释应是完整的句子。第一个单词应大写，除非它是小写字母开头的标识符（永远

不要更改标识符的大小写!)。

块注释通常由一个或多个由完整句子构成的段落组成，每句以句号结束。

在多句注释中，句号后应使用一到两个空格，除了最后一句。

确保你的注释清晰且易于被使用你编写语言的其他使用者理解。

来自非英语国家的 Python 编码者：请用英语编写注释，除非你 120% 确定代码永远不会被不讲你语言的人阅读。

7.1 块注释

块注释通常适用于其后的某些（或全部）代码，并与该代码缩进到同一级别。块注释的每一行以 # 和单个空格开头（除非是注释内的缩进文本）。

块注释中的段落由仅包含单个 # 的行分隔。

7.2 行内注释

谨慎使用行内注释。

行内注释是与语句在同一行的注释。行内注释应与语句至少间隔两个空格。它们应以 # 和单个空格开头。

如果行内注释陈述了显而易见的内容，则是不必要的，甚至会分散注意力。不要这样做:

```
# 错误:
x = x + 1                 # 增加 x
```

但有时候，这是有用的:

```
# 正确:
x = x + 1                 # 补偿边框
```

7.3 文档字符串

编写良好文档字符串（即 “docstrings”）的规范在 PEP 257 中得以永存。

- 为所有公共模块、函数、类和方法编写文档字符串。对于非公共方法，文档字符串不是必需的，但应有一个描述方法功能的注释。该注释应出现在 `def` 行之后。
- PEP 257 描述了良好的文档字符串规范。注意，最重要的是，结束多行文档字符串的 """ 应独占一行：

```
# 正确：

"""返回一个 foobang

可选的 plotz 表示首先对 bizbaz 进行 frobnicate。
"""
```

- 对于单行文档字符串，请将结束的 """ 保留在同一行：

```
# 正确：

"""返回一个 ex-parrot。"""
```

8 命名规范

Python 库的命名规范有些混乱，因此我们永远无法完全一致——尽管如此，以下是当前推荐的命名标准。新模块和包（包括第三方框架）应遵循这些标准，但在已有不同风格的库中，内部一致性优先。

8.1 覆盖原则

作为公共 API 的一部分对用户可见的名称应遵循反映使用而非实现的惯例。

8.2 描述性：命名风格

有许多不同的命名风格。能够识别使用的是哪种命名风格，有助于理解它们的作用。

以下是常用的命名风格：

- `b`（单个小写字母）
- `B`（单个大写字母）

- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords`（或 CapWords，或 CamelCase——因其字母的“驼峰”外观而得名 [4]）。有时也称为 StudlyCaps。

  注意：在 CapWords 中使用缩写时，将缩写的所有字母大写。因此，`HTTPServerError` 优于 `HttpServerError`。
- `mixedCase`（与 CapitalizedWords 不同的是以小写字母开头！）
- `Capitalized_Words_With_Underscores`（不美观！）

还有一种风格是使用短的唯一前缀来分组相关名称。在 Python 中不常用，但为完整起见提及。例如，`os.stat()` 函数返回的元组中的项传统上具有像 `st_mode`、`st_size`、`st_mtime` 等的名称。（这样做是为了强调与 POSIX 系统调用结构的字段对应关系，帮助熟悉该系统的程序员。）

X11 库对其所有公共函数使用前导 X。在 Python 中，这种风格通常被认为是不必要的，因为属性和方法名称以对象为前缀，函数名称以模块名称为前缀。

此外，以下使用前导或尾随下划线的特殊形式被识别（这些通常可以与任何大小写惯例结合使用）：

- `_single_leading_underscore`：弱“内部使用”指示符。例如，`from M import *` 不会导入以单个下划线开头的名称。
- `single_trailing_underscore_`：按惯例用于避免与 Python 关键字冲突，例如：

  ```
  # 正确：
  tkinter.Toplevel(master, class_='ClassName')
  ```

- `__double_leading_underscore`：在命名类属性时，触发名称改编（在类 FooBar 中，`__boo` 变为 `_FooBar__boo`；见下文）。

- `__double_leading_and_trailing_underscore__`: “魔法” 对象或属性，存在于用户控制的命名空间中。例如，`__init__`、`__import__` 或 `__file__`。永远不要发明这样的名称；仅按文档使用它们。

8.3 规定性：命名规范

8.3.1 要避免的名称

永远不要使用字符 ‘l’（小写字母 el）、‘O’（大写字母 oh）或 ‘I’（大写字母 eye）作为单字符变量名称。

在某些字体中，这些字符与数字 1 和 0 无法区分。如果想使用 ‘l’，请使用 ‘L’ 代替。

8.3.2 ASCII 兼容性

标准库中使用的标识符必须与 PEP 3131 的政策部分[5] 中描述的 ASCII 兼容。

8.3.3 包和模块名称

模块应使用短小的全小写名称。如果提高可读性，可以在模块名称中使用下划线。Python 包也应使用短小的全小写名称，尽管不鼓励使用下划线。

当用 C 或 C++ 编写的扩展模块有配套的 Python 模块提供更高级别（例如更面向对象）的接口时，C/C++ 模块应使用前导下划线（例如 `_socket`）。

8.3.4 类名称

类名称通常应使用 CapWords 惯例。

在接口被记录并主要用作可调用对象的情况下，可以使用函数的命名惯例。

注意，内置名称有单独的惯例：大多数内置名称是单个单词（或两个单词连在一起），CapWords 惯例仅用于异常名称和内置常量。

[5] https://peps.python.org/pep-3131/#policy-specification

8.3.5 类型变量名称

PEP 484[6] 中引入的类型变量名称通常应使用 CapWords 惯例，偏好短名称：`T`、`AnyStr`、`Num`。建议对用于声明协变或逆变行为的变量分别添加后缀 `_co` 或 `_contra`：

```
# 正确：
from typing import TypeVar

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

8.3.6 异常名称

因为异常应是类，所以这里适用类命名惯例。然而，如果异常确实是错误，应在异常名称上使用后缀 “Error”。

8.3.7 全局变量名称

（希望这些变量仅在一个模块内使用。）其惯例与函数的命名规则大致相同。

设计为通过 `from M import *` 使用的模块应使用 `__all__` 机制防止导出全局变量，或使用较旧的惯例，为此类全局变量添加前导下划线（你可能希望这样做以表明这些全局变量是 “模块非公开” 的）。

8.3.8 函数和变量名称

函数名称应为小写，必要时用下划线分隔单词以提高可读性。

变量名称遵循与函数名称相同的惯例。

仅在已普遍采用 mixedCase 的上下文中（例如 threading.py）允许使用 mixedCase，以保持向后兼容性。

8.3.9 函数和方法参数

始终对实例方法的第一个参数使用 `self`。

[6] https://peps.python.org/pep-0484/

始终对类方法的第一个参数使用 `cls`。

如果函数参数名称与保留关键字冲突，通常最好在参数名称后附加单个尾随下划线，而不是使用缩写或拼写错误。例如，`class_` 优于 `clss`。（或许更好的方法是通过使用同义词避免此类冲突。）

8.3.10 方法名称和实例变量

使用函数命名规则：小写，必要时用下划线分隔单词以提高可读性。

仅对非公开方法和实例变量使用单个前导下划线。

为避免与子类的名称冲突，使用两个前导下划线触发 Python 的名称改编规则。

Python 会将这些名称与类名称改编：如果类 Foo 有属性 `__a`，则无法通过 `Foo.__a` 访问。（执着的用户仍可通过调用 `Foo._Foo__a` 访问。）通常，仅在设计用于子类化的类中为避免属性名称冲突才应使用双前导下划线。

注意：关于 ___names 的使用存在一些争议（见下文）。

8.3.11 常量

常量通常在模块级别定义，全部大写，用下划线分隔单词。例如，`MAX_OVERFLOW` 和 `TOTAL`。

8.3.12 继承设计

始终决定类的哪些方法和实例变量（统称为 “属性”）应是公开还是非公开。如有疑问，选择非公开；稍后将其公开比将公开属性变为非公开更容易。

公开属性是你期望与你的类无关的客户端使用的属性，你承诺避免向后不兼容的更改。非公开属性是不打算由第三方使用的属性；你不保证非公开属性不会更改或甚至被移除。

我们在这里不使用 “私有” 一词，因为在 Python 中没有属性是真正私有的（除非付出通常不必要的努力）。

另一类属性是 “子类 API”（在其他语言中通常称为 “受保护”）。某些类设计为可被继承，以扩展或修改类行为。当设计此类类时，需明确决定哪些属性是公开的，哪些是子类 API 的一部分，哪些是仅由你的基类使用的。

考虑到这一点，以下是 Python 的指南：

- 公开属性不应有前导下划线。
- 如果你的公开属性名称与保留关键字冲突，请在属性名称后附加单个尾随下划线。这优于缩写或拼写错误。（然而，尽管有此规则，对于任何已知是类的变量或参数，尤其是类方法的第一个参数，首选拼写为 'cls'。）

  注意 1：见上文关于类方法参数名称的建议。
- 对于简单的公开数据属性，最好仅暴露属性名称，不使用复杂的访问器/修改器方法。请记住，Python 提供了未来增强的简单途径，如果发现简单的数据属性需要增加功能行为。在这种情况下，使用属性隐藏功能实现背后的简单数据属性访问语法。

  注意 1：尽量保持功能行为无副作用，尽管缓存等副作用通常是可以的。

  注意 2：避免对计算昂贵的操作使用属性；属性表示法让调用者认为访问（相对）廉价。
- 如果你的类打算被子类化，且你有不希望子类使用的属性，考虑使用双前导下划线且无尾随下划线命名。这会触发 Python 的名称改编算法，将类名称改编到属性名称中。这有助于避免子类意外包含同名属性导致的名称冲突。

  注意 1：注意，改编名称仅使用简单的类名称，因此如果子类选择了相同的类名和属性名，仍可能发生名称冲突。

  注意 2：名称改编可能使某些用途（如调试和 `__getattr__()`）不太方便。然而，名称改编算法文档完善，手动执行也很容易。

  注意 3：并非所有人都喜欢名称改编。尽量平衡避免意外名称冲突与高级调用者可能使用的需求。

8.4 公共和内部接口

任何向后兼容性保证仅适用于公共接口。因此，用户能够清楚地区分公共接口和内部接口非常重要。

文档化的接口被视为公共接口，除非文档明确声明它们是临时或内部接口，免除通常的向后兼容性保证。所有未文档化的接口应假定为内部接口。

为更好地支持内省，模块应使用 `__all__` 属性明确声明其公共 API 中的名称。将 `__all__` 设置为空列表表示该模块没有公共 API。

即使正确设置了 `__all__`，内部接口（包、模块、类、函数、属性或其他名称）仍应以单个前导下划线为前缀。

如果任何包含命名空间（包、模块或类）被视为内部接口，则该接口也被视为内部接口。

导入的名称应始终视为实现细节。其他模块不得依赖对这些导入名称的间接访问，除非它们是包含模块 API 的明确文档化部分，例如 `os.path` 或包的 `__init__` 模块暴露子模块的功能。

9 编程建议

- 代码应以不损害其他 Python 实现（PyPy、Jython、IronPython、Cython、Psyco 等）的方式编写。

  例如，不要依赖 CPython 对 `a += b` 或 `a = a + b` 形式的字符串连接的高效实现。这种优化即使在 CPython 中也很脆弱（仅对某些类型有效），且在不使用引用计数的实现中完全不存在。在库的性能敏感部分，应使用 `''.join()` 形式。这将确保在各种实现中连接操作具有线性时间复杂度。

- 与单例（如 None）的比较应始终使用 `is` 或 `is not`，永远不要使用相等运算符。

  另外，当你真正想表达 `if x is not None` 时，注意不要写 `if x`——例如，测试默认值为 None 的变量或参数是否被设置为其他值。其他值可能是容器等类型，在布尔上下文中可能为假！

- 使用 `is not` 运算符而不是 `not ... is`。虽然这两种表达式在功能上相同，但前者更易读且更可取：

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

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

- 实现具有丰富比较的排序操作时，最好实现所有六个操作（`__eq__`、`__ne__`、`__lt__`、`__le__`、`__gt__`、`__ge__`），而不是依赖其他代码仅使用特定比较。

  为减少工作量，`functools.total_ordering()` 装饰器提供了生成缺失比较方法的工具。

PEP 207[7] 表明 Python 假设存在自反规则。因此，解释器可能将 `y > x` 与 `x < y` 交换，`y >= x` 与 `x <= y` 交换，并可能交换 `x == y` 和 `x != y` 的参数。`sort()` 和 `min()` 操作保证使用 `<` 运算符，`max()` 函数使用 `>` 运算符。然而，最好实现所有六个操作，以免在其他上下文中引起混淆。

- 始终使用 def 语句，而不是直接将 lambda 表达式绑定到标识符的赋值语句：

```
# 正确：
def f(x): return 2*x
```

```
# 错误：
f = lambda x: 2*x
```

第一种形式意味着生成的函数对象的名称明确为 'f' 而不是通用的 ''。这对回溯和字符串表示通常更有用。使用赋值语句消除了 lambda 表达式相对于显式 def 语句的唯一优势（即可以嵌入到较大表达式中）。

- 从 `Exception` 而不是 `BaseException` 派生异常。直接从 `BaseException` 继承是为几乎总是错误捕获的异常保留的。

  根据捕获异常的代码可能需要的区分来设计异常层次结构，而不是根据异常抛出的位置。旨在以编程方式回答 "出了什么问题？"，而不是仅声明 "发生了问题"（参见 PEP 3151[8] 中关于内置异常层次结构的经验教训）。

  这里适用类命名惯例，尽管如果异常是错误，应在异常类上添加后缀 "Error"。用于非本地流控制或其他信号形式的非错误异常无需特殊后缀。

- 适当地使用异常链。`raise X from Y` 应用于指示显式替换，而不丢失原始回溯。

  当故意替换内部异常（使用 `raise X from None`）时，确保将相关细节转移到新异常（例如，将 KeyError 转换为 AttributeError 时保留属性名称，或将原始异常的文本嵌入到新异常消息中）。

- 捕获异常时，尽可能明确指定异常，而不是使用最基本的 `except:` 子句：

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

[7] https://peps.python.org/pep-0207/
[8] https://peps.python.org/pep-3151/

最基本的 `except:` 子句会捕获 `SystemExit` 和 `KeyboardInterrupt` 异常，使通过 Control-C 中断程序变得更困难，并可能掩盖其他问题。如果你想捕获所有表示程序错误的异常，使用 `except Exception:`（最基本的 except 等同于 `except BaseException:`）。

最基本的 'except' 子句的良好经验法则是仅限于两种情况：

1. 如果异常处理程序将打印或记录回溯；至少用户会知道发生了错误。

2. 如果代码需要执行一些清理工作，但随后通过 `raise` 让异常向上传播。`try...finally` 是处理这种情况的更好方式。

- 捕获操作系统错误时，优先使用 Python 3.3 引入的显式异常层次结构，而不是内省 `errno` 值。
- 此外，对于所有 try/except 子句，将 `try` 子句限制在绝对必要的最少代码量。这再次避免掩盖错误：

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

```
# 错误：
try:
    # 过于宽泛！
    return handle_value(collection[key])
except KeyError:
    # 也会捕获 handle_value() 抛出的 KeyError
    return key_not_found(key)
```

- 当资源仅用于特定代码段时，使用 `with` 语句确保使用后及时可靠地清理。try/finally 语句也是可接受的。
- 每当上下文管理器执行除获取和释放资源之外的操作时，应通过单独的函数或方法调用它们：

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

```
# 错误：
```

```
with conn:
    do_stuff_in_transaction(conn)
```

后一个例子未提供任何信息表明 `__enter__` 和 `__exit__` 方法在事务后关闭连接之外还执行了其他操作。在这种情况下明确非常重要。

- 在返回语句中保持一致。函数中的所有返回语句要么都返回表达式，要么都不返回。如果任何返回语句返回表达式,则无值返回的返回语句应明确声明为 `return None`，并且在函数末尾（如果可达）应存在显式返回语句：

```
# 正确：

def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

```
# 错误：

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

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

- 使用 `''.startswith()` 和 `''.endswith()` 而不是字符串切片来检查前缀或后缀。

  startswith() 和 endswith() 更干净且不易出错：

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

```
# 错误：
if foo[:3] == 'bar':
```

- 对象类型比较应始终使用 `isinstance()` 而不是直接比较类型：

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

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

- 对于序列（字符串、列表、元组），利用空序列为假的事实：

```
# 正确：
if not seq:
if seq:
```

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

- 不要编写依赖于显著尾随空白的字符串字面量。此类尾随空白在视觉上无法区分，一些编辑器（或最近的 reindent.py）会修剪它们。
- 不要使用 == 将布尔值与 True 或 False 比较：

```
# 正确：
if greeting:
```

```
# 错误：
if greeting == True:
```

更糟：

```
# 错误：
if greeting is True:
```

- 不鼓励在 `try...finally` 的 finally 套件中使用流控制语句 `return`/`break`/`continue`，如果这些流控制语句会跳出 finally 套件。因为此类语句会隐式取消通过 finally 套件传播的任何活跃异常：

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

9.1 函数注解

随着 PEP 484 的接受，函数注解的风格规则已更改。

- 函数注解应使用 PEP 484 语法（前述部分中有一些关于注解格式的建议）。
- 之前在本 PEP 中推荐的注解风格实验不再鼓励。
- 然而，在标准库之外，现在鼓励在 PEP 484 规则内的实验。例如，用 PEP 484 风格类型注解标记大型第三方库或应用程序，审查添加这些注解的难易程度，并观察它们的存在是否提高代码可理解性。
- Python 标准库在采用此类注解时应保守，但允许在新代码和重大重构中使用它们。
- 对于想以不同方式使用函数注解的代码，建议在文件顶部附近添加以下形式的注释：

```
# 正确：
# type: ignore
```

  这会告诉类型检查器忽略所有注解。（在 PEP 484 中可以找到更细粒度的禁用类型检查器警告的方法。）
- 与 linter 一样，类型检查器是可选的独立工具。Python 解释器默认不应因类型检查发出任何消息，也不应根据注解更改其行为。
- 不想使用类型检查器的用户可以自由忽略它们。然而，预计第三方库包的用户可能希望对这些包运行类型检查器。为此，PEP 484 推荐使用存根文件：.pyi 文件，类型检查器优先读取这些文件而非对应的.py 文件。存根文件可以与库一起分发，或（经库作者许可）通过 typeshed 仓库 [5] 单独分发。

9.2 变量注解

PEP 526[9] 引入了变量注解。其风格建议与上文所述的函数注解类似：

- 模块级变量、类和实例变量以及局部变量的注解在冒号后应有一个空格。
- 冒号前不应有空格。
- 如果赋值有右值，等号两侧应各有一个空格：

[9] https://peps.python.org/pep-526/

```python
# 正确：

code: int

class Point:
    coords: Tuple[int, int]
    label: str = '<unknown>'
```

```python
# 错误：

code:int  # 冒号后无空格
code : int  # 冒号前有空格

class Test:
    result: int=0  # 等号周围无空格
```

- 虽然 PEP 526 在 Python 3.6 中被接受，但变量注解语法是所有 Python 版本的存根文件的首选语法（详见 PEP 484）。

参考文献

[1] G. van Rossum,《PEP 7 - Style Guide for C Code》. https://peps.python.org/pep-0007/, 2001 年.

[2] B. Warsaw,《Barry's GNU Mailman style guide》. https://barry.warsaw.us/software/STYLEGUIDE.txt.

[3] D. Knuth, *The TeXbook*. 页 195 and 196.

[4] Wikipedia, https://en.wikipedia.org/wiki/Camel_case.

[5] Typeshed, hhttps://github.com/python/typeshed.

版权

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

Last modified: 2025-04-04 00:19:04 GMT