PEP 8 – Python 代码风格指南

作者： Guido van Rossum <guido at python.org>、Barry Warsaw <barry at python.org>、Nick Coghlan <ncoghlan at gmail.com>

状态： 积极的
类型： 过程
创建： 2001 年 7 月 5 日
后历史： 2001年7月5日、2013年8月1日

▶ 目录

介绍

本文档给出了构成主要 Python 发行版中标准库的 Python 代码的编码约定。请参阅随附的信息 PEP，描述Python 的 C 实现中的 C 代码的样式指南。

本文档和PEP 257（文档字符串约定）改编自 Guido 的原始 Python 样式指南文章，并添加了 Barry 的样式指南[2]中的一些内容。

随着附加约定的确定以及过去的约定因语言本身的变化而变得过时，该风格指南随着时间的推移而不断发展。

许多项目都有自己的编码风格指南。如果发生任何冲突，此类特定于项目的指南优先适用于该项目。

愚蠢的一致性是小头脑的恶魔

Guido 的重要见解之一是，代码的读取次数远多于编写次数。此处提供的指南旨在提高代码的可读性并使其在各种 Python 代码中保持一致。正如PEP 20所说，“可读性很重要”。

风格指南是关于一致性的。与本风格指南的一致性很重要。项目内的一致性更为重要。一个模块或功能内的一致性是最重要的。

然而，知道何时要不一致——有时风格指南的建议并不适用。如有疑问，请运用您的最佳判断。查看其他示例并决定哪个看起来最好。请随时询问！

特别是：不要仅仅为了遵守此 PEP 而破坏向后兼容性！

忽略特定准则的其他一些充分理由：

1. 应用指南会降低代码的可读性，即使对于习惯阅读遵循此 PEP 的代码的人也是如此。
2. 与周围的代码保持一致，这些代码也破坏了它（可能是出于历史原因）——尽管这也是清理别人混乱的机会（以真正的 XP 风格）。

3. 因为相关代码早于指南的引入，并且没有其他原因需要修改该代码。
4. 当代码需要与不支持样式指南推荐的功能的旧版 Python 保持兼容时。

代码布局

缩进

每个缩进级别使用 4 个空格。

连续行应该使用 Python 的隐式行连接在圆括号、方括号和大括号内垂直对齐包裹的元素，或者使用*悬挂缩进* [1]。使用悬挂缩进时应考虑以下事项：第一行不应该有参数，并且应该使用进一步的缩进来清楚地将其自身区分为连续行：

```
  # Correct:

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

  # Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
  def long_function_name(
          var_one, var_two, var_three,
          var_four):
      print(var_one)

  # Hanging indents should add a level.
  foo = long_function_name(
      var_one, var_two,
      var_three, var_four)


  # Wrong:

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

  # Further indentation required as indentation is not distinguishable.
  def long_function_name(
      var_one, var_two, var_three,
      var_four):
      print(var_one)
```

对于连续行，4 空格规则是可选的。

选修的：

```
  # Hanging indents *may* be indented to other than 4 spaces.
  foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

当 - 语句的条件部分 `if` 足够长，需要跨多行编写时，值得注意的是，两个字符关键字（即 ）、单个 `if` 空格以及左括号的组合会为多行条件的后续行创建自然的 4 个空格缩进。这可能会与嵌套在 - 语句内的缩进代码套件产生视觉冲突 `if`，该代码套件自然也会缩进 4 个空格。对于如何（或是否）进一步在视觉上将这些条件行与 - 语句内的嵌套套件区分开来，此 PEP 没有采取明确的立场 `if` 。在这种情况下可接受的选择包括但不限于：

```
# No extra indentation.
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# Add a comment, which will provide some distinction in editors
# supporting syntax highlighting.
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()

# Add some extra indentation on the conditional continuation line.
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

（另请参阅下面关于是否在二元运算符之前或之后中断的讨论。）

多行结构上的右大括号/方括号/圆括号可以排列在列表最后一行的第一个非空白字符下方，如下所示：

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

或者它可以排列在开始多行结构的行的第一个字符下，如下所示：

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

制表符还是空格?

空格是首选的缩进方法。

制表符应仅用于与已使用制表符缩进的代码保持一致。

Python 不允许混合制表符和空格进行缩进。

最大线路长度

将所有行限制为最多 79 个字符。

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

限制所需的编辑器窗口宽度可以并排打开多个文件，并且在使用在相邻列中显示两个版本的代码审查工具时效果很好。

大多数工具中的默认包装会破坏代码的视觉结构，使其更难以理解。选择这些限制是为了避免在窗口宽度设置为 80 的编辑器中进行换行，即使该工具在换行时在最后一列中放置了标记符号。一些基于网络的工具可能根本不提供动态换行。

有些团队强烈喜欢更长的线路。对于专门或主要由可以就此问题达成一致的团队维护的代码，可以将行长度限制增加到 99 个字符，前提是注释和文档字符串仍以 72 个字符换行。

Python 标准库比较保守，要求将行数限制为 79 个字符（文档字符串/注释限制为 72 个）。

换行长行的首选方法是在圆括号、方括号和大括号内使用 Python 的隐式续行。通过将表达式括在括号中，可以将长行分成多行。应优先使用这些内容而不是使用反斜杠来继续行。

反斜杠有时仍然是合适的。例如，`with` 在 Python 3.10 之前，长的多语句不能使用隐式延续，因此在这种情况下反斜杠是可以接受的：

```
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())
```

（有关多行 if 语句缩进的进一步思考，请参阅前面关于多行 if`with` 语句的讨论。）

另一个这样的情况是`assert` 语句。

确保适当缩进续行。

应该在二元运算符之前还是之后换行?

几十年来，推荐的风格是在二元运算符之后中断。但这可能会以两种方式损害可读性：运算符往往会分散在屏幕上的不同列中，并且每个运算符都会从其操作数移到上一行。在这里，眼睛必须做额外的工作来分辨哪些项目被添加，哪些项目被减去：

```
# Wrong:
# operators sit far away from their operands
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

为了解决这个可读性问题，数学家和他们的出版商遵循相反的惯例。*Donald Knuth 在他的计算机和排版*系列中解释了传统规则：“虽然段落中的公式总是在二元运算和关系之后中断，但显示的公式总是在二元运算之前中断”[3]。

遵循数学传统通常会产生更具可读性的代码：

```
# Correct:
# easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

在 Python 代码中，只要约定在本地一致，就允许在二元运算符之前或之后中断。对于新代码，建议采用 Knuth 风格。

空行

用两个空行包围顶级函数和类定义。

类内的方法定义由一个空行包围。

可以（谨慎地）使用额外的空行来分隔相关函数组。在一堆相关的单行代码（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指示逻辑部分。

Python 接受 control-L（即 ^L）换页符作为空格；许多工具将这些字符视为页面分隔符，因此您可以使用它们来分隔文件的相关部分的页面。请注意，某些编辑器和基于 Web 的代码查看器可能无法将 control-L 识别为换页符，并会在其位置显示另一个字形。

源文件编码

核心 Python 发行版中的代码应始终使用 UTF-8，并且不应具有编码声明。

在标准库中，非 UTF-8 编码只能用于测试目的。谨慎使用非 ASCII 字符，最好仅用于表示地点和人名。如果使用非 ASCII 字符作为数据，请避免使用嘈杂的 Unicode 字符，例如 z̯̯͡a̧͎̺l̡͓̫g̹̲o̡̼̘ 和字节顺序标记。

Python 标准库中的所有标识符必须使用纯 ASCII 标识符，并且应尽可能使用英语单词（在许多情况下，使用非英语的缩写和技术术语）。

鼓励拥有全球受众的开源项目采取类似的政策。

进口

- 导入通常应该在单独的行上：

  ```
  # Correct:
  import os
  import sys


  # Wrong:
  import sys, os
  ```

  不过这样说也没关系：

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

- 导入始终放在文件的顶部，紧接在任何模块注释和文档字符串之后，以及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. 相关第三方进口。
  3. 本地应用程序/库特定的导入。

  您应该在每组导入之间放置一个空行。

- 建议绝对导入，因为如果导入系统配置不正确（例如当包内的目录以 结尾时），它们通常更具可读性并且往往表现更好（或至少给出更好的错误消息） `sys.path`：

  ```
  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，特别是在处理复杂的包布局时，使用绝对导入会不必要地冗长：

  ```
  from . import sibling
  from .sibling import example
  ```

  标准库代码应避免复杂的包布局并始终使用绝对导入。

- 当从包含类的模块导入类时，通常可以这样拼写：

  ```
  from myclass import MyClass
  from foo.bar.yourclass import YourClass
  ```

  如果这种拼写导致本地名称冲突，请明确拼写它们：

  ```
  import myclass
  import foo.bar.yourclass
  ```

  并使用“myclass.MyClass”和“foo.bar.yourclass.YourClass”。

- 应避免通配符导入 ( )，因为它们使命名空间中存在哪些名称变得不清楚，从而使读者和许多自动化工具感到困惑。通配符导入有一个合理的用例，即重新发布内部接口作为公共 API 的一部分（例如，使用可选加速器模块中的定义覆盖接口的纯 Python 实现，并且事先不知道将覆盖哪些定义）。`from <module> import *`

  当以这种方式重新发布名称时，以下有关公共和内部接口的准则仍然适用。

模块级别 Dunder 名称

模块级“dunders”（即带有两个前导和两个尾随下划线的名称），例如 `__all__`、`__author__`、`__version__` 等应放置在模块文档字符串之后，但在*除* 导入之外的任何导入语句之前。Python 要求 future-imports 必须出现在模块中除文档字符串之外的任何其他代码之前：`from __future__`

```
"""This is the example module.

This module does stuff.
"""

from __future__ import barry_as_FLUFL

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

import os
import sys
```

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 并未对此提出建议。选择一个规则并遵守它。但是，当字符串包含单引号或双引号字符时，请使用另一个以避免字符串中出现反斜杠。它提高了可读性。

对于三引号字符串，请始终使用双引号字符，以与PEP 257中的文档字符串约定保持一致。

表达式和语句中的空格

讨厌的事

在以下情况下避免使用多余的空格：

- 紧邻圆括号、方括号或大括号内：

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


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

- 在尾随逗号和后面的右括号之间：

```
# Correct:
foo = (0,)


# Wrong:
bar = (0, )
```

- 紧接在逗号、分号或冒号之前：

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


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

- 但是，在切片中，冒号的作用类似于二元运算符，并且两侧应具有相同的数量（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的间距。例外：当省略切片参数时，空格也被省略：

```
# Correct:
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]


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

- 紧接在开始函数调用的参数列表的左括号之前：

```
# Correct:
spam(1)


# Wrong:
spam (1)
```

- 紧接在开始索引或切片的左括号之前：

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


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

- 赋值（或其他）运算符周围有多个空格以将其与另一个运算符对齐：

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


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

其他建议

- 避免在任何地方出现尾随空格。因为它通常是不可见的，所以可能会令人困惑：例如，反斜杠后跟空格和换行符不算作行继续标记。一些编辑器不保留它，并且许多项目（例如 CPython 本身）都有拒绝它的预提交挂钩。
- 始终在这些二元运算符两侧添加一个空格：赋值（`=`）、扩展赋值（`+=` 等 `-=` ）、比较（`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, , , ），布尔值（, , ）。`not in` `is` `is not` `and` `or` `not`
- 如果使用具有不同优先级的运算符，请考虑在优先级最低的运算符周围添加空格。使用你自己的判断；但是，切勿使用多个空格，并且二元运算符两侧始终具有相同数量的空格：

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


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

- 函数注释应使用冒号的正常规则，并且箭头周围始终有空格（`->` 如果存在）。（ 有关函数注释的更多信息，请参阅下面的函数注释。）：

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


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

- = 当用于指示关键字参数或用于指示 *未注释的*函数参数的默认值时，不要在符号周围使用空格：

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


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

  但是，当将参数注释与默认值组合时，请在 = 符号周围使用空格：

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


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

- 通常不鼓励复合语句（同一行上的多个语句）：

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

  而不是：

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

- 虽然有时可以将 if/for/while 与一个小主体放在同一行，但切勿对多子句语句这样做。还要避免折叠这么长的线！

  而不是：

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

  当然不：

```
# Wrong:
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()
```

何时使用尾随逗号

尾随逗号通常是可选的，但在创建一个元素的元组时它们是强制性的。为了清楚起见，建议将后者括在（技术上多余的）括号中：

```
# Correct:
FILES = ('setup.cfg',)


# Wrong:
FILES = 'setup.cfg',
```

当尾随逗号是多余的时，当使用版本控制系统时，当值、参数或导入项的列表预计会随着时间的推移而扩展时，它们通常很有用。该模式是将每个值（等）单独放在一行上，始终添加尾随逗号，并在下一行添加右括号/方括号/大括号。然而，在与结束分隔符相同的行上使用尾随逗号是没有意义的（除了上述单例元组的情况）：

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


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

评论

与代码相矛盾的注释比没有注释更糟糕。当代码更改时，始终优先考虑使注释保持最新！

注释应该是完整的句子。第一个单词应大写，除非它是以小写字母开头的标识符（切勿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组成的段落组成，每个句子以句号结尾。

在多句注释中，除了最后一句之外，您应该在句末句号后使用一两个空格。

确保您的评论对于使用您所写语言的其他使用者来说清晰且易于理解。

来自非英语国家的 Python 程序员：请用英语写下您的注释，除非您 120% 确定该代码永远不会被不会说您的语言的人阅读。

阻止评论

块注释通常适用于其后面的部分（或全部）代码，并且缩进到与该代码相同的级别。块注释的每一行都以# 和一个空格开头（除非它是注释内的缩进文本）。

块注释内的段落由包含单个#.

内嵌评论

谨慎使用内联注释。

内联注释是与语句位于同一行的注释。内联注释与语句之间应至少用两个空格分隔。它们应该以 # 和一个空格开头。

内联注释是不必要的，而且如果它们陈述了显而易见的内容，实际上会分散注意力。不要这样做：

```
x = x + 1                 # Increment x
```

但有时，这很有用：

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

文档字符串

编写良好文档字符串（又名“文档字符串”）的约定在PEP 257中永垂不朽。

- 为所有公共模块、函数、类和方法编写文档字符串。对于非公共方法来说，文档字符串不是必需的，但您应该有一条注释来描述该方法的用途。此注释应出现在该 def 行之后。
- PEP 257描述了良好的文档字符串约定。请注意，最重要的是， """ 结束多行文档字符串的 应该单独占一行：

  ```
  """Return a foobang

  Optional plotz says to frobnicate the bizbaz first.
  """
  ```

- 对于单行文档字符串，请将结尾保持 """ 在同一行：

  ```
  """Return an ex-parrot."""
  ```

命名约定

Python 库的命名约定有点混乱，因此我们永远无法使其完全一致——尽管如此，以下是当前推荐的命名标准。新的模块和包（包括第三方框架）应按照这些标准编写，但如果现有库具有不同的风格，则首选内部一致性。

压倒一切的原则

作为 API 公共部分对用户可见的名称应遵循反映用法而不是实现的约定。

描述性：命名风格

有很多不同的命名风格。它有助于识别正在使用的命名样式，而与它们的用途无关。

通常区分以下命名风格：

- `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`（或者 CapWords，或者 CamelCase——如此命名是因为它的字母看起来凹凸不平[4]）。有时也称为 StudlyCaps。

  注意：在 CapWords 中使用缩略词时，请将缩略词的所有字母大写。因此 HTTPServerError 比 HttpServerError 更好。

- `mixedCase`（与 CapitalizedWords 的区别在于首字母小写字符！）
- `Capitalized_Words_With_Underscores`（丑陋的！）

还有一种使用简短的唯一前缀将相关名称分组在一起的风格。这在 Python 中用得不多，但为了完整性还是提到了。例如，该函数`os.stat()`返回一个元组，其项目传统上具有诸如`st_mode`、 等名称。（这样做是为了强调与 POSIX 系统调用结构体字段的对应关系，这有助于程序员熟悉这一点。）`st_size` `st_mtime`

X11 库在其所有公共函数中使用前导 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，还可以识别以下使用前导或尾随下划线的特殊形式（这些形式通常可以与任何大小写约定结合使用）：

- `_single_leading_underscore`：“内部使用”指标较弱。例如，不导入名称以下划线开头的对象。`from M import *`
- `single_trailing_underscore_`：按惯例使用以避免与Python关键字冲突，例如

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

- `__double_leading_underscore`：命名类属性时，调用名称修改（在类 FooBar 内，`__boo`变为`_FooBar__boo`；见下文）。
- `__double_leading_and_trailing_underscore__`：存在于用户控制的命名空间中的“神奇”对象或属性。例如`__init__`，`__import__`或`__file__`。永远不要发明这样的名字；仅按照记录使用它们。

规定性：命名约定

应避免使用的名字

切勿使用字符“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。当想要使用“l”时，请改用“L”。

ASCII 兼容性

标准库中使用的标识符必须与 ASCII 兼容，如 PEP 3131 的策略部分中所述。

包和模块名称

模块应该有短的、全小写的名称。如果可以提高可读性，可以在模块名称中使用下划线。Python 包也应该有短的、全小写的名称，尽管不鼓励使用下划线。

当用 C 或 C++ 编写的扩展模块具有提供更高级别（例如更面向对象）接口的随附 Python 模块时，C/C++ 模块具有前导下划线（例如 ） `_socket` 。

类名

类名通常应使用 CapWords 约定。

在接口被记录并主要用作可调用的情况下，可以使用函数的命名约定。

请注意，内置名称有一个单独的约定：大多数内置名称是单个单词（或两个单词一起运行），CapWords 约定仅用于异常名称和内置常量。

类型变量名称

PEP 484中引入的类型变量的名称通常应使用大写字母，优先使用短名称： `T` , `AnyStr` , `Num` 。建议在用于相应声明协变或逆变行为的变量中添加后缀 `_co` or： `_contra`

```
from typing import TypeVar

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

异常名称

因为异常应该是类，所以这里应用类命名约定。但是，您应该在异常名称上使用后缀“Error”（如果异常实际上是错误）。

全局变量名称

（我们希望这些变量只能在一个模块内使用。）这些约定与函数的约定大致相同。

设计用于 via 的模块应该使用该机制来防止导出全局变量，或者使用旧的约定，在此类全局变量前添加下划线前缀（您可能希望这样做以指示这些全局变量是“模块非公共”）。 `from M import *` `__all__`

函数和变量名称

函数名称应小写，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约定。

仅在已成为流行风格的上下文中（例如 threading.py）才允许使用混合大小写，以保持向后兼容性。

函数和方法参数

始终用于 `self` 实例方法的第一个参数。

始终用于 `cls` 类方法的第一个参数。

如果函数参数的名称与保留关键字冲突，通常最好附加一个尾随下划线，而不是使用缩写或拼写错误。因此 `class_` 优于 `clss`。（也许更好的方法是使用同义词来避免此类冲突。）

方法名称和实例变量

使用函数命名规则：小写，单词之间用下划线分隔，以提高可读性。

仅对非公共方法和实例变量使用一个前导下划线。

为了避免与子类发生名称冲突，请使用两个前导下划线来调用 Python 的名称修饰规则。

Python 将这些名称与类名混合在一起：如果类 Foo 有一个名为 的属性 `__a`，则无法通过 访问它 `Foo.__a`。（坚持不懈的用户仍然可以通过调用 来获得访问权限 `Foo._Foo__a`。）通常，双前导下划线应该仅用于避免与设计为子类化的类中的属性发生名称冲突。

注意：关于 __names 的使用存在一些争议（见下文）。

常数

常量通常在模块级别定义，并全部用大写字母书写，并用下划线分隔单词。示例包括 `MAX_OVERFLOW` 和 `TOTAL`。

为继承而设计

始终决定类的方法和实例变量（统称为“属性”）应该是公共的还是非公共的。如有疑问，请选择非公开；稍后将其公开比将公共属性设为非公开更容易。

公共属性是您期望与您的类无关的客户端使用的属性，并承诺避免向后不兼容的更改。非公共属性是指那些不打算被第三方使用的属性；您不保证非公共属性不会更改甚至被删除。

我们在这里不使用术语“私有”，因为在 Python 中没有属性是真正私有的（通常不需要做不必要的工作）。

另一类属性是“子类 API”的一部分（在其他语言中通常称为“受保护”）。有些类被设计为继承、扩展或修改类行为的各个方面。在设计这样的类时，请注意明确决定哪些属性是公共的、哪些是子类 API 的一部分以及哪些确实只能由基类使用。

考虑到这一点，以下是 Pythonic 指南：

- 公共属性不应有前导下划线。
- 如果您的公共属性名称与保留关键字冲突，请在您的属性名称后附加一个尾随下划线。这比缩写或损坏的拼写更好。（然而，尽管有这条规则，“cls”是任何已知为类的变量或参数的首选拼写，尤其是类方法的第一个参数。）

  注 1：有关类方法，请参阅上面的参数名称建议。

- 对于简单的公共数据属性，最好只公开属性名称，而不需要复杂的访问器/修改器方法。请记住，如果您发现简单的数据属性需要扩展功能行为，Python 提供了一条未来增强的简单路径。在这种情况下，请使用属性将功能实现隐藏在简单的数据属性访问语法后面。

注 1：尽量保持功能行为没有副作用，尽管缓存等副作用通常是好的。

注2：避免使用属性进行计算量大的操作；属性表示法使调用者相信访问（相对）便宜。

- 如果您的类打算进行子类化，并且您有不希望子类使用的属性，请考虑使用双前导下划线命名它们，并且不使用尾随下划线。这会调用 Python 的名称修饰算法，其中类的名称被修饰为属性名称。如果子类无意中包含具有相同名称的属性，这有助于避免属性名称冲突。

  注意1：请注意，重整名称中仅使用简单的类名称，因此如果子类选择相同的类名称和属性名称，仍然可能会出现名称冲突。

  注 2：名称修改可能会使某些用途（例如调试）变得 `__getattr__()` 不太方便。然而，名称修饰算法有详细记录并且易于手动执行。

  注 3：并不是每个人都喜欢名称修改。尝试在避免意外名称冲突的需要与高级调用者的潜在使用之间取得平衡。

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。

记录的接口被认为是公共的，除非文档明确声明它们是临时或内部接口，不受通常的向后兼容性保证。所有未记录的接口都应假定为内部接口。

为了更好地支持自省，模块应使用属性在其公共 API 中显式声明名称 `__all__` 。设置 `__all__` 为空列表表示该模块没有公共 API。

即使 `__all__` 设置正确，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）被视为内部的，则接口也被视为内部的。

导入的名称应始终被视为实现细节。其他模块不得依赖于对此类导入名称的间接访问，除非它们是包含模块的 API 的显式记录部分，例如公开子模块功能的 `os.path` 包模块。 `__init__`

编程建议

- 代码的编写方式不应损害 Python 的其他实现（PyPy、Jython、IronPython、Cython、Psyco 等）。

  例如，不要依赖 CPython 对或形式的语句进行就地字符串连接的高效实现。即使在 CPython 中，这种优化也很脆弱（它只适用于某些类型），并且在不使用引用计数的实现中根本不存在。在库的性能敏感部分，应改用表单。这将确保在各种实现中串联在线性时间内发生。 `a += b` `a = a + b` `''.join()`

- 与像 None 这样的单例的比较应该总是使用 `is` or来完成，而不是使用相等运算符。 `is not`

  另外，当你真正想写的时候要小心——例如，当测试默认为 None 的变量或参数是否被设置为其他值时。另一个值可能具有在布尔上下文中可能为 false 的类型（例如容器）！ `if x` `if x is not None`

- 使用运算符而不是. 虽然这两个表达式在功能上相同，但前者更具可读性和首选： `is not` `not ... is`

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


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

- 当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执行特定比较。

  为了最大限度地减少所涉及的工作量， `functools.total_ordering()` 装饰器提供了一个工具来生成缺失的比较方法。

  PEP 207表明Python 假定了*自反性规则*。因此，解释器可以与、 和交换，并且可以交换和的参数。and运算保证使用运算符，而函数则使用 运算符。但是，最好实现所有六个操作，以便在其他上下文中不会出现混乱。`y > x` `x < y` `y >= x` `x <= y` `x == y` `x != y` `sort()` `min()` `<` `max()` `>`

- 始终使用 def 语句而不是将 lambda 表达式直接绑定到标识符的赋值语句：

  ```
  # Correct:
  def f(x): return 2*x


  # Wrong:
  f = lambda x: 2*x
  ```

  第一种形式意味着生成的函数对象的名称具体为“f”，而不是通用的“<lambda>”。一般来说，这对于回溯和字符串表示更有用。赋值语句的使用消除了 lambda 表达式相对于显式 def 语句所能提供的唯一好处（即它可以嵌入到更大的表达式中）

- `Exception` 从而不是派生异常 `BaseException` 。直接继承 from `BaseException` 是为异常保留的，在这些异常中捕获它们几乎总是错误的做法。

  *根据捕获*异常的代码可能需要的区别（而不是引发异常的位置）来设计异常层次结构 。旨在回答“出了什么问题？”的问题。以编程方式，而不是仅仅声明“发生了问题”（有关内置异常层次结构的本课示例，请参阅PEP 3151 ）

  类命名约定适用于此，但如果异常是错误，则应将后缀“Error”添加到异常类中。用于非本地流控制或其他形式的信令的非错误异常不需要特殊后缀。

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

  当有意替换内部异常（使用）时，请确保相关详细信息已转移到新异常（例如在将 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）。 `raise X from None`

- 捕获异常时，请尽可能提及特定异常，而不是使用裸 `except:` 子句：

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

  裸 `except:` 子句将捕获 SystemExit 和 KeyboardInterrupt 异常，从而使使用 Control-C 中断程序变得更加困难，并且可能掩盖其他问题。如果要捕获表示程序错误的所有异常，请使用 (bare except 相当于)。 `except Exception:` `except BaseException:`

  一个好的经验法则是将纯粹的“例外”子句的使用限制为两种情况：

  1. 异常处理程序是否将打印或记录回溯；至少用户会意识到发生了错误。
  2. 如果代码需要执行一些清理工作，但随后让异常向上传播 `raise` 。 `try...finally` 可能是处理这种情况的更好方法。
- 捕获操作系统错误时，更喜欢 Python 3.3 中引入的显式异常层次结构，而不是 `errno` 值的内省。
- 此外，对于所有 try/ except 子句，将该 `try` 子句限制为所需的绝对最小代码量。同样，这可以避免掩盖错误：

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


# Wrong:
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

- 当资源位于特定代码段的本地时，请使用 `with` 语句来确保其在使用后得到及时可靠的清理。try/finally 语句也是可以接受的。
- 每当上下文管理器执行获取和释放资源以外的操作时，都应通过单独的函数或方法来调用它们：

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


# Wrong:
with conn:
    do_stuff_in_transaction(conn)
```

后一个示例没有提供任何信息来表明 和 `__enter__` 方法 `__exit__` 除了在事务后关闭连接之外正在执行其他操作。在这种情况下，明确很重要。

- 返回语句保持一致。函数中的所有 return 语句要么都应该返回表达式，要么都不应该返回。如果任何 return 语句返回一个表达式，则任何不返回值的 return 语句都应将其显式声明为，并且显式 return 语句应出现在函数末尾（如果可到达）： `return None`

```
# Correct:

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)


# Wrong:

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

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

- 使用 ''.startswith() and ''.endswith() 代替字符串切片来检查前缀或后缀。

  startswith() 和endswith() 更干净并且更不容易出错：

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


  # Wrong:
  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：

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


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

- 对于序列（字符串、列表、元组），请使用空序列为 false 的事实：

  ```
  # Correct:
  if not seq:
  if seq:


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

- 不要编写依赖于重要尾随空格的字符串文字。这种尾随空白在视觉上是无法区分的，一些编辑器（或者最近的 reindent.py）会修剪它们。
- 不要使用以下方法将布尔值与 True 或 False 进行比较 == ：

  ```
  # Correct:
  if greeting:


  # Wrong:
  if greeting == True:
  ```

  更差：

  ```
  # Wrong:
  if greeting is True:
  ```

- return //不鼓励 在 a 的finally 套件中使用流程控制语句，其中流程控制语句会跳转 break 到finally 套件之外。这是因为这样的语句将隐式取消通过finally套件传播的任何活动异常： continue try...finally

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

函数注释

随着PEP 484的接受，函数注释的样式规则发生了变化。

- 函数注释应使用PEP 484语法（上一节中有一些注释的格式建议）。
- 不再鼓励对本 PEP 中之前推荐的注释样式进行实验。
- 然而，在 stdlib 之外， 现在鼓励在PEP 484规则范围内进行实验。例如，使用PEP 484样式类型注释标记大型第三方库或应用程序，检查添加这些注释的容易程度，并观察它们的存在是否会提高代码的可理解性。
- Python 标准库在采用此类注释时应该保守，但它们的使用是允许用于新代码和大型重构的。
- 对于想要以不同方式使用函数注释的代码，建议添加以下形式的注释：

  ```
  # type: ignore
  ```

  靠近文件顶部；这告诉类型检查器忽略所有注释。（可以在PEP 484中找到禁用类型检查器投诉的更细粒度的方法。）

- 与 linter 一样，类型检查器是可选的独立工具。默认情况下，Python 解释器不应因类型检查而发出任何消息，也不应根据注释改变其行为。
- 不想使用类型检查器的用户可以忽略它们。但是，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建议使用存根文件：类型检查器优先于相应的 .py 文件读取的 .pyi 文件。存根文件可以与库一起分发，也可以通过 typeshed repo [5]单独分发（在库作者的许可下）。

变量注释

PEP 526引入了变量注释。对它们的风格建议与上面描述的函数注释的风格建议类似：

- 模块级变量、类和实例变量以及局部变量的注释应在冒号后有一个空格。
- 冒号之前不应有空格。
- 如果赋值有右侧，则等号两侧应恰好有一个空格：

  ```
  # Correct:

  code: int

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


  # Wrong:

  code:int  # No space after colon
  code : int  # Space before colon

  class Test:
      result: int=0  # No spaces around equality sign
  ```

- 尽管Python 3.6 接受PEP 526 ，但变量注释语法是所有 Python 版本上存根文件的首选语法（有关详细信息，请参阅 PEP 484 ）。

脚注

[1]

*悬挂缩进*是一种排版样式，段落中除第一行外的所有行均缩进。在 Python 上下文中，该术语用于描述一种样式，其中带括号的语句的左括号是该行的最后一个非空白字符，后续行缩进直到右括号。

参考

[2]
Barry 的 GNU Mailman 风格指南 http://barry.warsaw.us/software/STYLEGUIDE.txt

[3]
Donald Knuth 的*The TeXBook*，第 195 和 196 页。

[4]
http://www.wikipedia.com/wiki/CamelCase

[5]
Typeshed 仓库 https://github.com/python/typeshed

版权

来源：https ://github.com/python/peps/blob/main/pep-0008.txt

最后修改: 2023-04-30 14:23:54+00:00 GMT