PEP 8 – Python代码样式指南

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*

类别: *过程*

创建于: 2001年7月5日

提交历史: 2001年7月5日，2013年8月1日

简介

本文档给出了包含主要 Python 发行版中标准库的 Python 代码的编码约定。请参阅描述Python C 实现中 C 代码样式指南的配套信息 PEP 。

本文档和PEP 257（文档字符串约定）改编自 Guido 的原始 Python 风格指南文章，并从 Barry 的风格指南[1]中添加了一些内容。

这个风格指南随着时间的推移而发展，因为附加的约定被识别出来，过去的约定因语言本身的变化而变得过时。

许多项目都有自己的编码风格指南。 如有任何冲突，此类特定于项目的指南优先于本项目的使用。

愚蠢的一致性是小脑袋的妖精

Guido 的主要见解之一是{**代码被阅读的频率远高于其编写的频率**}。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。与本风格指南保持一致很重要。**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。**

但是，知道什么时候不一致——有时风格指南的建议并不适用。如有疑问，请使用您的最佳判断。查看其他示例并确定最好的示例。不要犹豫，问！

特别是：*不要为了遵守此 PEP 而破坏向后兼容性！*

忽略特定指南的其他一些充分理由：

1. 应用指南会使代码的可读性降低，即使对于习惯阅读遵循此 PEP 的代码的人也是如此。
2. 为了与也会破坏它的周围代码保持一致（可能是出于历史原因）——尽管这也是一个清理别人的烂摊子的机会（以真正的 XP 风格）。
3. 因为有问题的代码早于指南的引入，并且没有其他理由修改该代码。
4. 当代码需要与不支持样式指南推荐的功能的旧版本 Python 保持兼容时。

代码布局

缩进

每个缩进级别使用 4 个空格。

续行应该使用 Python 的隐式行在圆括号、方括号和大括号内连接，或者使用悬挂缩进 [2]来垂直对齐包装元素。使用悬挂缩进时，应考虑以下因素；第一行不应该有任何参数，并且应该使用进一步的缩进来清楚地将自己区分为续行：

=== "正确的"

```python
# 与左括号对齐
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）以将参数与其他参数区分开来。
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 悬垂缩进应该增加一个层次。
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

=== "错误的"

```
``` python
# 不使用垂直对齐时禁止第一行的参数。
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 由于缩进不可区分，因此需要进一步缩进。
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```
```

4个空格规则对于续行是可选的。

可选的:

```
# 悬挂缩进 *可以* 缩进比到 4 个空格更少的位置。
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

当 `if` 语句的条件部分足够长，需要跨多行编写时，值得注意的是，两个字符关键字（即 `if` ）、`一个空格` 和 `一个左括号` 的组合会创建一个自然的 多行条件的后续行的 `4` 个空格缩进。这可能会与嵌套在 `if` 语句中的缩进代码套件产生视觉冲突，它自然也会缩进 `4` 个空格。 该 PEP 没有明确说明如何（或是否）在视觉上进一步区分此类条件行与 `if` 语句内的嵌套套件。这种情况下可接受的选择包括但不限于：

```
# 没有额外的缩进。
if (this_is_one_thing and
    that_is_another_thing):
    do_something()

# 添加注释，这将在支持语法突出显示的编辑器中提供一些区别。
if (this_is_one_thing and
    that_is_another_thing):
    # 既然这两个条件都成立，我们就可以 frobnicate 了。
    do_something()

# 在条件延续行上添加一些额外的缩进。
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：

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]

result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

或者它可能排在开始多行构造的行的第一个字符下，如：

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

制表符或空格？

空格是首选的缩进方法。

制表符应仅用于与已经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。

最大行长

将所有行限制为最多 **79** 个字符。

对于具有较少结构限制（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封装破坏了代码的视觉结构，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，即使该工具在换行时将标记符号放置在最后一列中也是如此。一些基于网络的工具可能根本不提供动态换行。

一些团队非常喜欢更长的线路长度。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 标准库是保守的，要求将行限制为 **79** 个字符（文档字符串/注释限制为 **72** 个）。

封装长行的首选方法是在`圆括号`、`方括号`和`大括号`内使用 Python 的隐含行延续。通过将表达式括在括号中，可以将长行分成多行。应该优先使用这些而不是使用反斜杠来续行。

反斜杠有时可能仍然适用。例如，`with` 在 `Python 3.10` 之前，长的多语句不能使用隐式延续，因此在这种情况下可以接受反斜杠：

```
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())
```

（请参阅前面关于多行 if 语句{target="_blank"}的讨论，以进一步了解此类多行`with`语句的缩进。）

另一个这样的例子是`assert`语句。

确保适当地缩进续行。

换行应该在二元运算符之前还是之后？

几十年来，推荐的风格是在二元运算符之后中断。但这会以两种方式损害可读性：运算符往往分散在屏幕上的不同列中，并且每个运算符都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，哪些被减去：

```
# 错误的:
# 运算符远离他们的操作数
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：

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

在 Python 代码中，允许在二元运算符之前或之后中断，只要约定在本地保持一致即可。对于新代码，建议使用 *Knuth* 的风格。

空行

{**用两个空行包围顶级函数和类定义。**}

{**类中的方法定义由一个空行包围。**}

可以（有节制地）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。

谨慎地在函数中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，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`，并且不应有编码声明。

在标准库中，非 `UTF-8` 编码应仅用于测试目的。谨慎使用非 `ASCII` 字符，最好仅用于表示地名和人名。如果使用非 `ASCII` 字符作为数据，请避免嘈杂的 `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，并且在可行的情况下应该使用**英语单词**（在许多情况下，使用的缩写词和技术术语不是英语）。

鼓励具有全球受众的开源项目采用类似的政策。

导入

- 导入模块通常应该在不同的行：
  === "正确的"

  ```
  ```python
  import os
  import sys
  ```
  ```

  === "错误的"

  ```
  ```python
  import sys, os
  ```
  ```

  可以这样说：

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

- 导入总是放在文件的顶部，就在**任何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. 相关的第三方进口。
  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**。
- 建议使用**绝对导入**，因为如果导入系统配置不正确（例如当包内的目录结束时），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，例如：`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。
- 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
  ```

  如果此拼写导致本地名称冲突，请明确拼写：

  ```
  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 (*)，因为它们会使命名空间中存在哪些名称变得不清楚，从而混淆读者和许多自动化工具。通配符导入有一个可靠的用例，它是将内部接口重新发布为公共 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，以及确切的定义是什么）事先不知道被覆

盖）。`from <module> import *`
以这种方式重新发布名称时，以下关于公共和内部接口的指南仍然适用。

模块级 Dunder 名称

模块级别的“dunders”（即带有两个前导和两个尾随下划线的名称），例如`__all__`，`__author__`，`__version__`等应该放在模块文档字符串之后但除了 导入之外的任何导入语句之前。Python 要求 `future-imports` 必须出现在除文档字符串之外的任何其他代码之前的模块中：`from __future__`

```
"""这是一个示例模块

这个模块不做任何事
"""

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 不对此提出建议。选择一个规则并坚持下去。但是，当字符串包含单引号或双引号字符时，请使用另一个来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，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

烦恼

在以下情况下避免多余的空格：

- 紧接在圆括号、方括号或大括号内：

  === "正确的"

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

  === "错误的"

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

- 在尾随逗号和紧随其后的右括号之间：

  === "正确的"

  ```python
  foo = (0,)
  ```

  === "错误的"

  ```python
  bar = (0, )
  ```

- 紧接在逗号、分号或冒号之前：

  === "正确的"

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

  === "错误的"

  ```python
  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，并且两边的数量应该相等（将其视为具有最低优先级的运算符）。在扩展切片中，两个冒号必须应用相同的间距。例外：省略切片参数时，省略空格：

  === "正确的"

```python
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]
```

=== "错误的"

```python
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

- 紧接在开始函数调用的参数列表的左括号之前：

  === "正确的"

  ```python
  spam(1)
  ```

  === "错误的"

  ```python
  spam (1)
  ```

- 在开始索引或切片的左括号之前：

  === "正确的"

  ```python
  dct['key'] = lst[index]
  ```

  === "错误的"

  ```python
  dct ['key'] = lst [index]
  ```

- 赋值（或其他）运算符周围有多个空格以使其与另一个运算符对齐：

=== "正确的"

```python
x = 1
y = 2
long_variable = 3
```

=== "错误的"

```python
x             = 1
y             = 2
long_variable = 3
```

其他建议

- 避免在任何地方尾随空格。因为它通常是不可见的，所以可能会造成混淆：例如反斜杠后跟空格和换行符不算作续行标记。一些编辑器不保留它，许多项目（如 CPython 本身）都有拒绝它的预提交挂钩。
- 始终在两边用一个空格包围这些二元运算符：赋值（`=`）、扩充赋值（`+=` 等 `-=`）、比较（`==`，`<`，`>`，`!=`，`<>`，`<=`，`>=`，`in`，`not in`，`is`，`is not`）、布尔值（`and`，`or`，`not`）。
- 如果使用具有不同优先级的运算符，根据你自己的判断请考虑在具有**最低优先级的运算符周围添加空格**；但是，永远不要使用超过一个空格，并且在二元运算符的两边总是有相同数量的空格：

  === "正确的"

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

  === "错误的"

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

- 函数注释应该使用冒号的正常规则，并且 `->` 如果存在的话，箭头周围总是有空格。（ 有关函数注释的更多信息，请参见下面的函数注释。）：
  === "正确的"

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

  === "错误的"

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

- 当 `=` 用于指示关**键字参数**或用于指示 **未注释函数参数的默认值时**，请勿在符号周围使用空格：
  === "正确的"

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

  === "错误的"

  ```python
  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：
  === "正确的"

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

=== "错误的"

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

- 通常不鼓励使用复合语句（同一行上的多个语句）：

  === "正确的"

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

  === "错误的"

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

- 虽然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。还要避免折叠这么长的线！

  === "正确的"

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

  === "错误的"

```python
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，除非在制作一个元素的元组时它们是强制性的。为清楚起见，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，它们通常在使用版本控制系统时有用，当值列表、参数或导入项预计会随着时间的推移而扩展时。该模式是将每个值（等）单独放在一行，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，在结束定界符所在的同一行上有一个尾随逗号是没有意义的（除了上述单例元组的情况）：

=== "正确的"

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

=== "错误的"

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

注释

与代码相矛盾的注释比没有注释更糟糕。当代码更改时，始终优先保持注释是最新的！

注释应该是完整的句子。第一个单词应该大写，除非它是以小写字母开头的标识符（永远不要改变标识符的大小写！）。

块注释通常由一个或多个段落组成，这些段落由完整的句子组成，每个句子以句号结尾。

在多语句注释中，你应该在句末句号之后使用一到两个空格，但在最后一句话之后除外。

确保您的注释清晰易懂，其他使用您所用语言的人也易于理解。

来自非英语国家的 Python 程序员：请用英语写你的注释，除非你 120% 确定代码永远不会被不会说你的语言的人阅读。

块级注释

块级注释通常适用于它们后面的部分（或全部）代码，并缩进到与该代码相同的级别。块级注释的每一行都以一个 `#` 空格开头（除非它是注释中的缩进文本）。

块级注释内的段落由包含单个 `#` 分隔.

行级注释

谨慎使用行内注释。

行内注释是与语句位于同一行的注释。行内注释应与语句至少用两个空格分隔。它们应该以 `#` 和一个空格开头。

行内注释是不必要的，如果它们陈述显而易见的话，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
```

文档注释

编写好的文档注释（又名“`docstrings`”）的约定在PEP 257中永垂不朽。

- 为所有`公共模块`、`函数`、`类`和`方法`编写文档注释。非公共方法不需要`Docstrings`，但您应该有一个注释来描述该方法的作用。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`“””`应该单独成行：

  ```
  """Return a foobang

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

- 对于一个单行文档注释，请将结尾放在`"""`同一行

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

命名约定

Python 库的命名约定有点乱，所以我们永远无法做到完全一致——尽管如此，这里是目前推荐的命名标准。新的模块和包（包括第三方框架）应该按照这些标准编写，但是如果现有的库有不同的风格，内部一致性是首选。

至高无上的原则

作为 API 的公共部分对用户可见的名称应**遵循实际使用**而非实现的约定。

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风格，独立于它们的用途。

通常区分以下命名样式：

- `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[^4]的凹凸外观而得名）。这有时也称为 `StudlyCaps`。
  注意：在 `CapWords` 中使用首字母缩略词时，将首字母缩写词的所有字母大写。因此 `HTTPServerError` 优于 `HttpServerError`。
- `mixedCase`（与 `CapitalizedWords` 的首字母小写字符不同！）
- `Capitalized_Words_With_Underscores`（丑陋的！）

还有使用简短的唯一前缀将相关名称组合在一起的风格。 这在 `Python` 中用得不多，但为了完整起见提到了它。 例如，`os.stat()` 函数返回一个元组，其项通常具有 `st_mode`、`st_size`、`st_mtime` 等名称。（这样做是为了强调与 `POSIX` 系统调用结构的字段的对应关系，这有助于程序员熟悉它。）

`X11` 库为其所有公共函数使用前导 `X`。 在 Python 中，这种样式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，而函数名称以模块名称为前缀。

此外，还识别以下使用前导或尾随下划线的特殊形式（这些通常可以与任何大小写约定结合使用）：

- `_single_leading_underscore`：弱的“**内部使用**”指标。例如，不导入名称以下划线开头的对象。`from M import *`
- `single_trailing_underscore_`：按照惯例使用以避免与 `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“**魔法**”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。

规范：命名约定

要避免的名称

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。

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是 `ASCII` 兼容的，如 PEP 3131 的政策部分所述。

包和模块名称

模块应该有简短的全小写名称。如果提高可读性，可以在模块名称中使用下划线。`Python` 包也应该有简短的、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）接口的附带 `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。

在记录接口并主要用作可调用接口的情况下，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。

输入变量名

PEP 484中引入的类型变量的名称通常应该使用 `CapWords`，更喜欢短名称：`T`, `AnyStr`, `Num`。建议为用于声明 `协变` 或 `逆变行为` 的变量添加后缀 `_co` 或者 `_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，所以这里适用类命名约定。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

（希望这些变量仅供在一个模块内使用。）这些约定与函数的约定大致相同。

设计为通过 `from M import *` 使用的模块应该使用 `__all__` 机制来防止导出全局变量，或者使用在此类全局变量前加上下划线的旧约定（您可能想这样做以表明这些全局变量是“**模块的而非公共**”）。

函数和变量名称

{**函数名应该是小写的，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。

`mixedCase` 只允许在已经是流行风格的上下文中使用（例如 `threading.py`），以保持向后兼容性。

函数和方法参数

始终使用 `self` 作为实例方法的第一个参数。

始终使用 `cls` 作为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，通常最好在末尾附加一个下划线，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：**小写字母，必要时用下划线分隔单词以提高可读性。**

仅对非公共方法和实例变量使用一个前导下划线。

为避免与子类的名称冲突，请使用两个前导下划线来调用 Python 的名称修改规则。

Python 将这些名称与类名称混淆：如果类 `Foo` 具有名为 `__a` 的属性，则 `Foo.__a` 无法访问它。（坚持不懈的用户仍然可以通过调用 `Foo._Foo__a` 获得访问权限。）通常，双前导下划

线应该只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）。

常量

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

始终决定一个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，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。

公共属性是您希望您的类的不相关客户使用的属性，并且您承诺避免向后不兼容的更改。 非公共属性是指不打算由第三方使用的属性； 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“ `private` ”，因为在 Python 中没有属性是真正私有的（通常没有不必要的工作量）。

另一类属性是那些属于“subclass API”（在其他语言中通常称为“ `protected` (受保护)”）的属性。某些类被设计为继承自，以扩展或修改类行为的各个方面。在设计这样一个类时，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，哪些确实只能由您的基类使用。

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部下划线。这比缩写或损坏的拼写更可取。（然而，尽管有这条规则，' `cls` ' 是已知为类的任何变量或参数的首选拼写，尤其是类方法的第一个参数。）
  注 1：请参阅上面关于类方法的参数名称建议。
- 对于简单的公共数据属性，最好只公开属性名称，而不要使用复杂的访问器/修改器方法。请记住，如果您发现一个简单的数据属性需要增加功能行为，Python 为未来的增强提供了一条简单的途径。在这种情况下，使用属性将功能实现隐藏在简单的数据属性访问语法之后。
  注 1：尽管缓存等副作用通常没有问题，但尽量保持功能行为无副作用。
  注 2：避免将属性用于计算量大的操作； 属性表示法使调用者相信访问（相对的）便宜。
- 如果您的类打算被子类化，并且您有不希望子类使用的属性，请考虑使用双前导下

划线和没有尾随下划线的方式命名它们。这会调用 `Python` 的名称修改算法，其中类的名称被修改为属性名称。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 1：请注意，在经过修饰的名称中只使用了简单的类名，因此如果子类选择了相同的类名和属性名，您仍然会遇到名称冲突。
注 2: 名称修改可以使某些用途（例如调试和 `__getattr__()`）不太方便。 然而，名称修改算法有据可查，并且易于手动执行。
注 3：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界面和内部界面是很重要的。

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的或内部接口，不受通常的向后兼容性保证的约束。**应假定所有未记录的接口都是内部的。**

为了更好地支持内省，模块应该使用属性在其公共 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__` 设置得当，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作为前缀。

如果任何包含的命名空间（包、模块或类）被认为是内部的，那么接口也被认为是内部的。

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 API 的明确记录部分，例如 `os.path` 或包的 `__init__` 模块公开子模块的功能。

编程建议

- 代码的编写方式应不损害 `Python` 的其他实现（`PyPy`、`Jython`、`IronPython`、`Cython`、`Psyco` 等）。
  例如，不要依赖 `CPython` 对 `a += b` 或 `a = a + b` 形式的语句的就地字符串连接的高效实现。 这种优化即使在 `CPython` 中也是脆弱的（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，应该使用

`''.join()` 形式。这将确保跨各种实现在线性时间内发生串联。

- 与像 `None` 这样的单例比较应该总是用 `is` 或 `is not` 来完成，永远不要用相等运算符(`==`)。
  另外，当你真正的意思是 `if x` 不是 `None` 时，请注意写 `if x` ——例如 在测试默认为 `None` 的变量或参数是否设置为其他值时。 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）！
- 使用 `is not` 运算符而不是 `not ... is`。 虽然这两个表达式在功能上是相同的，但前者更具可读性和首选：

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

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

- 在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。
  为了尽量减少所涉及的工作， `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 表明自反性规则由 `Python` 假定。 因此，解释器可以交换 `y > x` 和 `x < y` ， `y >= x` 和 `x <= y` ，并且可以交换 `x == y` 和 `x != y` 的参数。 `sort()` 和 `min()` 操作保证使用 `<` 运算符，而 `max()` 函数使用 `>` 运算符。 但是，最好实现所有六个操作，以免在其他上下文中出现混淆。
- 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特别是“ `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）
- 从 `Exception` 而不是 `BaseException` 派生异常。 从 `BaseException` 直接继承是为异常保留的，在这些异常中捕获它们几乎总是错误的。
  根据捕获异常的代码可能需要的区别设计异常层次结构，而不是根据引发异常的位

置。 旨在回答“出了什么问题？”这个问题。 以编程方式，而不是仅声明“发生问题”（有关为内置异常层次结构学习的本课示例，请参阅 PEP 3151）
类命名约定适用于此处，但如果异常是错误，则应在异常类中添加后缀“`Error`”。 用于非本地流量控制或其他形式的信号的非错误异常不需要特殊后缀。

- 适当地使用异常链接。 `raise X from Y` 应该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时（使用 `raise X from None`），确保将相关细节转移到新的异常中（例如在将 `KeyError` 转换为 `AttributeError` 时保留属性名称，或者在新的异常消息中嵌入原始异常的文本）.
- 捕获异常时，尽可能提及特定异常，而不是使用简单的 `except:` 子句：

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

  一个简单的 `except:` 子句将捕获 `SystemExit` 和 `KeyboardInterrupt` 异常，使使用 `Control-C` 中断程序变得更加困难，并且可以掩盖其他问题。 如果要捕获所有表示程序错误的异常，请使用 `except Exception:`（纯 `except` 等同于 `except BaseException:`）。
  一个好的经验法则是将裸“`except`”子句的使用限制在两种情况下：
  1. 如果异常处理程序将打印出或记录回溯； 至少用户会意识到发生了错误。
  2. 如果代码需要做一些清理工作，然后让异常通过 `raise` 向上传播。 `try...finally` 可能是处理这种情况的更好方法。

函数注解

变量注解

版权

来源：https://github.com/python/peps/blob/main/pep-0008.txt{target="_blank"}

最后修改时间：2023-02-25 12:44:10 GMT

参考

1. Barry 的 GNU Mailman 风格指南 http://barry.warsaw.us/software/STYLEGUIDE.txt{target="_blank"} ↩
2. 悬挂缩进是一种排版样式，其中段落中除第一行外的所有行都缩进。在 Python 的上下文中，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。 ↩
3. Donald Knuth 的 《The TeXBook》，第 195 和 196 页。 ↩