Python PEP-8 编码风格指南中文版

2019-06-10

PEP 8 – Python 编码风格

PEP:	8
Title:	Style Guide for Python Code
Author:	Guido van Rossum <guido at python.org>, Barry Warsaw <barry at python.org>, Nick Coghlan <ncoghlan at gmail.com>
Status:	Active
Type:	Process
Created:	05-Jul-2001
Post-History:	05-Jul-2001, 01-Aug-2013

介绍 (Introduction)

这篇文档说明了 Python 主要发行版中标准库代码所遵守的规范。对于 Python 的 C 语言实现中的编码规范，请参考实现 Python 的 C 代码风格指南PEP 7。

这篇文档和PEP 257(Docstring 约定) 都改编自 Guido[1]最早的 Python 风格指南文章，并且添加了一些 Barry 的风格指南[2]中的内容。

语言自身在发生着改变，随着新的规范的出现和旧规范的过时，代码风格也会随着时间演变。

很多项目都有自己的一套风格指南。若和本指南有任何冲突，应该优先考虑其项目相关的那套指南。

[1] Python 之父

[2] Barry 的 GNU Mailman 编码风格指南

保持盲目的一致是头脑简单的表现 (A Foolish Consistency Is The Hobgoblin Of Little Minds)[3]

Guido 的一个重要观点是代码被读的次数远多于被写的次数。这篇指南旨在提高代码的可读性，使浩瀚如烟的 Python 代码风格能保持一致。正如PEP 20那首《Zen of Python》的小诗里所说的："可读性很重要 (Readability counts)"。

这本风格指南是关于一致性的。同风格指南保持一致性是重要的，但是同项目保持一致性更加重要，同一个模块和一个函数保持一致性则最为重要。

然而最最重要的是：要知道何时去违反一致性，因为有时风格指南并不适用。当存有疑虑时，请自行做出最佳判断。请参考别的例子去做出最好的决定。并且不要犹豫，尽管提问。

特别的：千万不要为了遵守这篇 PEP 而破坏向后兼容性！

[3]出自 Ralph Waldo Emerson。

例外

如果有以下借口，则可以忽略这份风格指南：

1. 当采用风格指南时会让代码更难读，甚至对于习惯阅读遵循这篇 PEP 的代码的人来说也是如此。
2. 需要和周围的代码保持一致性，但这些代码违反了指南中的风格（可是时历史原因造成的）——尽管这可能也是一个收拾别人烂摊子的机会（进入真正的极限编程状态）。
3. 若是有问题的某段代码早于引入指南的时间，那么没有必要去修改这段代码。
4. 代码需要和更旧版本的 Python 保持兼容，而旧版本的 Python 不支持风格指南所推荐的特性。

代码布局 (Code Lay-Out)

缩进 (Indentation)

每个缩进级别采用 4 个空格。

连续行所包装的元素应该要么采用 Python 隐式续行，即垂直对齐于圆括号、方括号和花括号，要么采用悬挂缩进 (hanging indent)[4]。采用悬挂缩进时需考虑以下两点：第一行不应该包括参数，并且在续行中需要再缩进一级以便清楚表示。

[4]挂起缩进是一种类型设置样式，除了第一行外，段落中的所有行都缩进。在 Python 的上下文中，该术语用于描述括号括起来的语句的左括号是该行的最后一个非空格字符的样式，后面的行将缩进直到右括号。

正确的例子:

```
# 同开始分界符 (左括号) 对齐
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 续行多缩进一级以同其他代码区别
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 悬挂缩进需要多缩进一级
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

错误的例子:

```
# 采用垂直对齐时第一行不应该有参数
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 续行并没有被区分开，因此需要再缩进一级
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

对于续行来说，4 空格的规则可以不遵守。

同样可行的例子:

```
# 悬挂缩进可以不采用 4 空格的缩进方法。
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

如果 `if` 语句太长，需要用多行书写，2 个字符 (例如,`if`) 加上一个空格和一个左括号刚好是 4 空格的缩进，但这对多行条件语句的续行是没用的。因为这会和 `if` 语句中嵌套的其他的缩进的语句产生视觉上的冲突。这份 PEP 中并没有做出明确的说明应该怎样来区分条件语句和 `if` 语句中所嵌套的语句。以下几种方法都是可行的，但不仅仅只限于这几种方法：

```
# 不采用额外缩进
if (this_is_one_thing and
    that_is_another_thing):
    do_something()
```

```python
# 增加一行注释，在编辑器中显示时能有所区分
# supporting syntax highlighting.
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()

# 在条件语句的续行增加一级缩进
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

多行结束右圆/方/花括号可以单独一行书写，和上一行的缩进对齐：

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

也可以和多行开始的第一行的第一个字符对齐：

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

Tab 还是空格？(Tab Or Space?)

推荐使用空格来进行缩进。

Tab 应该只在现有代码已经使用 tab 进行缩进的情况下使用，以便和现有代码保持一致。

Python 3 不允许 tab 和空格混合使用。

Python 2 的代码若有 tab 和空格混合使用的情况，应该把 tab 全部转换为只有空格。

当使用命令行运行 Python 2 时，使用`-t` 选项，会出现非法混用 tab 和空格的警告。当使用`-tt` 选项时，这些警告会变成错误。强烈推荐使用这些选项！

每行最大长度 (Maximum Line Length)

- 将所有行都限制在 79 个字符长度以内。
- 对于连续大段的文字（比如文档字符串 (docstring) 或注释），其结构上的限制更少，这些行应该被限制在 72 个字符长度内。
- 限制编辑器的窗口宽度能让好几个文件同时打开在屏幕上显示，在使用代码评审 (code review) 工具时在两个相邻窗口显示两个版本的代码效果很好。
- 很多工具的默认自动换行会破坏代码的结构，使代码更难以理解。在窗口大小设置为 80 个字符的编辑器中，即使在换行时编辑器可能会在最后一列放置一个记号，为避免自动换行也需要限制每行字符长度。一些基于 web 的工具可能根本没有自动换行的功能。
- 一些团队会强烈希望行长度比 79 个字符更长。当代码仅仅只由一个团队维护时，可以达成一致让行长度增加到 80 到 100 字符 (实际上最大行长是 99 字符)，注释和文档字符串仍然是以 72 字符换行。
- Python 标准库比较传统，将行长限制在 79 个字符以内（文档字符串/注释为 72 个字符）。

一种推荐的换行方式是利用 Python 圆括号、方括号和花括号中的隐式续行。长行可以通过在括号内换行来分成多行。应该最好加上反斜杠来区别续行。

有时续行只能使用反斜杠。例如，较长的多个 `with` 语句不能采用隐式续行，只能接受反斜杠表示换行：

```
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())
```

（参照前面关于多行 if 语句的讨论来进一步考虑这里 `with` 语句的缩进。）

另一个这样的例子是 `assert` 语句。

要确保续行的缩进适当。

二元运算符之前还是之后换行？(Should a line break before or after a binary operator?)

长期以来一直推荐的风格是在二元运算符之后换行。但是这样会影响代码可读性，包括两个方面：一是运算符会分散在屏幕上的不同列上，二是每个运算符会留在前一行并远离操作数。所以，阅读代码的时候眼睛必须做更多的工作来确定哪些操作数被加，哪些操作数被减：

```
# 错误的例子：运算符远离操作数
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

为了解决这个可读性问题，数学家及其出版商遵循相反的规定。Donald Knuth 在他的“电脑和排版”系列中解释了传统的规则：“尽管在段落中的公式总是在二元运算符之后换行，但显示公式时总是在二元运算符之前换行”[5]。

```
# 正确的例子：更容易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，只要在统一项目中约定一致，就可以在二元运算符之前或之后换行。对于新编写的代码，建议使用 Knuth 的风格。

[5] Donald Knuth 的《The TeXBook》，第 195 和 196 页。

空行 (Blank Line)

使用 2 个空行来分隔最外层的函数 (function) 和类 (class) 定义。

使用 1 个空行来分隔类中的方法 (method) 定义。

可以使用额外的空行（尽量少）来分隔一组相关的函数。在一系列相关的仅占一行的函数之间，空行也可以被省略（比如一组虚函数定义）。

在函数内使用空行（尽量少）使代码逻辑更清晰。

Python 支持 control-L（如:^L）换页符作为空格；许多工具将这些符号作为分页符，因此你可以使用这些符号来分页或者区分文件中的相关区域。注意，一些编辑器和基于 web 的代码预览器可能不会将 control-L 识别为分页符，而是显示成其他符号。

源文件编码 (Source File Encoding)

Python 核心发行版中的代码应该一直使用 UTF-8（Python 2 中使用 ASCII）。
使用 ASCII（Python 2）或者 UTF-8（Python 3）的文件不应该添加编码声明。
在标准库中，只有用作测试目的，或者注释或文档字符串需要提及作者名字而不得不使用非 ASCII 字符时，才能使用非默认的编码。否则，在字符串文字中包括非 ASCII 数据时，推荐使用`\x`, `\u`, `\U` 或`\N` 等转义符。
对于 Python 3.0 及其以后的版本中，标准库遵循以下原则（参见PEP 3131）：Python 标准库中的所有标识符都**必须**只采用 ASCII 编码的标识符，在可行的条件下也**应当**使用英文词（很多情况下，使用的缩写和技术术语词都不是英文）。此外，字符串文字和注释应该只包括 ASCII 编码。只有两种例外：

- (a) 测试情况下为了测试非 ASCII 编码的特性
- (b) 作者名字。作者名字不是由拉丁字母组成的也**必须**提供一个拉丁音译名。

鼓励具有全球受众的开放源码项目采用类似的原则。

模块引用 (Imports)

- Imports 应该分行写，而不是都写在一行，例如：

```
# 分开写
import os
import sys

# 不要像下面一样写在一行
import sys, os
```

这样写也是可以的：

```
from subprocess import Popen, PIPE
```

- Imports 应该写在代码文件的开头，位于模块 (module) 注释和文档字符串 (docstring) 之后，模块全局变量 (globals) 和常量 (constants) 声明之前。

  Imports 应该按照下面的顺序分组来写：

  1. 标准库 imports
  2. 相关第三方 imports
  3. 本地应用/库的特定 imports

  不同组的 imports 之前用空格隔开。

- 推荐使用绝对 (absolute)imports，因为这样通常更易读，在 import 系统没有正确配置（比如中的路径以 `sys.path` 结束）的情况下，也会有更好的表现（或者至少会给出错误信息）：

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

然而，除了绝对 imports，显式的相对 imports 也是一种可以接受的替代方式。特别是当处理复杂的包布局 (package layouts) 时，采用绝对 imports 会显得啰嗦。

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

标准库代码应当一直使用绝对 imports，避免复杂的包布局。

隐式的相对 imports 应该**永不**使用，并且 Python 3 中已经被去掉了。

- 当从一个包括类的模块中 import 一个类时，通常可以这样写：

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

如果和本地命名的拼写产生了冲突，应当直接 import 模块：

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

然后使用"myclass.MyClass" 和"foo.bar.yourclass.YourClass"。

- 避免使用通配符 imports(`from <module> import *`)，因为会造成在当前命名空间出现的命名含义不清晰，给读者和许多自动化工具造成困扰。有一个可以正当使用通配符 import 的情形，即将一个内部接口重新发布成公共 API 的一部分（比如，使用备选的加速模块中的定义去覆盖纯 Python 实现的接口，预先无法知晓具体哪些定义将被覆盖）。

  当使用这种方式重新发布命名时，指南后面关于公共和内部接口的部分仍然适用。

模块级的双下划线命名 (Module level dunder names)

模块中的“双下滑线”（变量名以两个下划线开头，两个下划线结尾）变量，比如 `__all__`，`__author`，`__version__` 等，应该写在文档字符串 (docstring) 之后，除了 `form __future__` 引用 (imports) 的任何其它类型的引用语句之前。

Python 要求模块中 `__future__` 的导入必须出现在除文档字符串 (docstring) 之外的任何其他代码之前。

例如：

```
"""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
```

字符串引用 (String Quotes)

在 Python 中表示字符串时，不管用单引号还是双引号都是一样的。但是不推荐将这两种方式看作一样并且混用。最好选择一种规则并坚持使用。当字符串中包含单引号时，采用双引号来表示字符串，反之也是一样，这样可以避免使用反斜杠，代码也更易读。

对于三引号表示的字符串，使用双引号字符来表示[6]，这样可以和PEP 257的文档字符串（docstring）规则保持一致。

[6]即用""" 而不是'''

表达式和语句中的空格 (Whitespace In Expressions And Statements)

一些痛点 (Pet Peeves)

在下列情形中避免使用过多的空白：

- 方括号，圆括号和花括号之后：

```
# 正确的例子：
spam(ham[1], {eggs: 2})

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

- 逗号，分号或冒号之前：

```
# 正确的例子：
if x == 4: print x, y; x, y = y, x

# 错误的例子：
if x == 4 : print x , y ; x , y = y , x
```

- 不过，在切片操作时，冒号和二元运算符是一样的，应该在其左右两边保留相同数量的空格（就像对待优先级最低的运算符一样）。在扩展切片操作中，所有冒号的左右两边空格数都应该相等。不过也有例外，当切片操作中的参数被省略时，应该也忽略空格。

```
# 正确的例子:
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]

# 错误的例子:
ham[lower + offset:upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

- 在调用函数时传递参数 list 的括号之前：

```
# 正确的例子:
spam(1)

# 错误的例子:
pam (1)
```

- 在索引和切片操作的左括号之前：

```
# 正确的例子:
dct['key'] = lst[index]

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

- 赋值 (或其他) 运算符周围使用多个空格来和其他语句对齐：

```
# 正确的例子:
x = 1
y = 2
long_variable = 3

# 错误的例子:
x             = 1
y             = 2
long_variable = 3
```

其它建议 (Other Recommendations)

- 避免任何行末的空格。因为它通常是不可见的, 它可能会令人困惑：例如反斜杠后跟空格和换行符不会作为续行标记。一些编辑器会自动去除行末空格, 许多项目（如 CPython 本身）都有提交前的预处理钩子来自动去除行末空格。

- 在二元运算符的两边都使用一个空格：赋值运算符 (=), 增量赋值运算符 (+=, -= etc.), 比较运算符 (==, <, >, !=, <>, <=, >=, in, not in, is, is not), 布尔运算符 (and, or, not)。

- 如果使用了优先级不同的运算符，则在优先级较低的操作符周围增加空白。请你自行判断，不过永远不要用超过 1 个空格，永远保持二元运算符两侧的空白数量一样。

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

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

- 使用 = 符号来表示关键字参数或参数默认值时，不要在其周围使用空格。

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

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

- 函数注解中的：也遵循一般的：加空格的规则，在-> 两侧各使用一个空格。(参见函数注解)

```
# 正确的例子：
def munge(input: AnyStr): ...
def munge() -> AnyStr: ...

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

- 在组合使用函数注解和参数默认值时，需要在 = 两侧各使用一个空格（只有当这个参数既有函数注解，又有默认值的时候）。

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

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

- 复合语句（即将多行语句写在一行）一般是不鼓励使用的。

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

# 最好不要这样:
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

- 有时也可以将短小的 if/for/while 中的语句写在一行，但对于有多个分句的语句永远不要这样做。也要避免将多行都写在一起。

```
# 最好不要这样：
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()

# 绝对不要这样：
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)
```

何时在末尾加逗号 (When to use trailing commas)

末尾逗号通常是可选的，除非在定义单元素元组 (tuple) 时是必需的（而且在 Python 2 中，它们具有 print 语句的语义）。为了清楚起见，建议使用括号（技术上来说是冗余的）括起来。

```
# 正确的例子：
FILES = ('setup.cfg',)

# 也正确，但令人困惑：
FILES = 'setup.cfg',
```

当使用版本控制系统时，在将来有可能扩展的列表末尾添加冗余的逗号是有好处的。具体的做法是将每一个元素写在单独的一行，并在行尾添加逗号，右括号单独占一行。但是，与有括号在同一行的末尾元素后面加逗号是没有意义的（上述的单元素元组除外）。

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

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

注释 (Comments)

和代码矛盾的注释还不如没有。当代码有改动时，一定要优先更改注释使其保持最新。

注释应该是完整的多个句子。如果注释是一个短语或一个句子，其首字母应该大写，除非开头是一个以小写字母开头的标识符（永远不要更改标识符的大小写）。

如果注释很短，结束的句号可以被忽略。块注释通常由一段或几段完整的句子组成，每个句子都应该以句号结束。

你应该在句尾的句号后再加上 2 个空格。

使用英文写作，参考 Strunk 和 White 的《The Elements of Style》

来自非英语国家的 Python 程序员们，请使用英文来写注释，除非你 120% 确定你的代码永远不会被不懂你所用语言的人阅读到。

块注释（Block Comments）

块注释一般写在对应代码之前，并且和对应代码有同样的缩进级别。块注释的每一行都应该以 `#` 和一个空格开头（除非该文本是在注释内缩进对齐的）。

块注释中的段落应该用只含有单个 `#` 的一行隔开。

行内注释（Inline Comments）

尽量少用行内注释。

行内注释是和代码语句写在一行内的注释。行内注释应该至少和代码语句之间有两个空格的间隔，并且以 `#` 和一个空格开始。

行内注释通常不是必要的，在代码含义很明显时甚至会让人分心。请不要这样做：

```
x = x + 1                 # x 自加
```

但这样做是有用的：

```
x = x + 1                 # 边界补偿
```

文档字符串 (Documentation Strings)

要知道如何写出好的文档字符串（docstring），请参考PEP 257。

- 所有的公共模块，函数，类和方法都应该有文档字符串。对于非公共方法，文档字符串不是必要的，但你应该留有注释说明该方法的功能，该注释应当出现在 `def` 的下一行。
- PEP 257描述了好的文档字符串应该遵循的规则。其中最重要的是，多行文档字符串以单行`"""` 结尾，不能有其他字符，例如：

```
"""Return a foobang

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

- 对于仅有一行的文档字符串，结尾处的`"""` 应该也写在这一行。

命名约定（Naming Conventions）

Python 标准库的命名约定有一些混乱，因此我们永远都无法保持一致。但如今仍然存在一些推荐的命名标准。新的模块和包（包括第三方框架）应该采用这些标准，但若是已经存在的包有另一套风格的话，还是应当与原有的风格保持内部一致。

首要原则（Overriding Principle）

对于用户可见的公共部分 API，其命名应当表达出功能用途而不是其具体的实现细节。

描述：命名风格（Descriptive: Naming Styles）

存在很多不同的命名风格，最好能够独立地从命名对象的用途认出采用了哪种命名风格。

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`(小写)
- `lower_case_with_underscores`(带下划线小写)
- `UPPERCASE`(大写)
- `UPPER_CASE_WITH_UNDERSCORES`(带下划线大写)
- `CapitalizedWords` (也叫做 CapWords 或者 CamelCase – 因为单词首字母大写看起来很像驼峰)。也被称作 StudlyCaps。

  注意：当 CapWords 里包含缩写时，将缩写部分的字母都大写。`HTTPServerError` 比 `HttpServerError` 要好。
- `mixedCase` (注意：和 CapitalizedWords 不同在于其首字母小写！)
- `Capitalized_Words_With_Underscores` (这种风格超丑！)

也有风格使用简短唯一的前缀来表示一组相关的命名。这在 Python 中并不常见，但为了完整起见这里也捎带提一下。比如，`os.stat()` 函数返回一个 tuple，其中的元素名原本为 `st_mode`,`st-size`,`st_mtime` 等等。（这样做是为了强调和 POSIX 系统调用结构之间的关系，可以让程序员更熟悉。）

X11 库中的公共函数名都以 X 开头。在 Python 中这样的风格一般被认为是不必要的，因为属性和方法名之前已经有了对象名的前缀，而函数名前也有了模块名的前缀。

此外，要区别以下划线开始或结尾的特殊形式（可以和其它的规则结合起来）：

- `_single_leading_underscore`: 以单个下划线开头是“内部使用”的弱标志。比如，`from M import *` 不会 import 下划线开头的对象。
- `single_trailing_underscore_`: 以单个下划线结尾用来避免和 Python 关键词产生冲突，例如:

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

- `__double_leading_underscore`: 以双下划线开头的风格命名类属性表示触发命名修饰（在 FooBar 类中，`__boo` 命名会被修饰成 `_FooBar__boo`; 见下）。
- `__double_leading_and_trailing_underscore__`: 以双下划线开头和结尾的命名风格表示“魔术”对象或属性，存在于用户控制的命名空间 (user-controlled namespaces) 里（也就是说，这些命名已经存在，但通常需要用户覆写以实现用户所需要的功能）。比如，`__init__`，`__import__` 或 `__file__`。请依照文档描述来使用这些命名，千万不要自己发明。

规范：命名约定 (Prescriptive: Naming Conventions)

需要避免的命名 (Names To Avoid)

不要使用字符'l'（L 的小写的字母），'O'（o 大写的字母），或者'I'（i 的大写的字母）来作为单个字符的变量名。

在一些字体中，这些字符和数字 1 和 0 无法区别开来。比如，当想使用'l' 时，使用'L' 代替。

ASCII 兼容性 (ASCII Compatibility)

标准库中使用的标识符必须与 ASCII 兼容 (参见PEP 3131中的policy这一节) 。

包和模块命名 (Package And Module Names)

模块命名应短小，且为全小写。若下划线能提高可读性，也可以在模块名中使用。Python 包命名也应该短小，且为全小写，但不应使用下划线。

当使用 C 或 C++ 写的扩展模块有相应的 Python 模块提供更高级的接口时（比如，更加面向对象），C/C++ 模块名以下划线开头（例如，`_sociket`）。

类命名 (Class Names)

类命名应该使用驼峰（CapWords）的命名约定。

当接口已有文档说明且主要是被用作调用时，也可以使用函数的命名约定。

注意对于内建命名 (builtin names) 有一个特殊的约定：大部分内建名都是一个单词（或者两个一起使用的单词），驼峰 (CapWords) 的约定只对异常命名和内建常量使用。

类型变量命名 (Type variable names)

PEP 484中引入的类型变量名称通常应使用简短的驼峰命名：`T`，`AnyStr`，`Num`。建议将后缀 `_co` 或 `_contra` 添加到用于声明相应的协变 (covariant) 和逆变 (contravariant) 的行为。例如：

```
from typing import TypeVar

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

异常命名 (Exception Names)

由于异常实际上也是类，因此类命名约定也适用与异常。不同的是，如果异常实际上是抛出错误的话，异常名前应该加上"Error" 的前缀。

全局变量命名 (Global Variable Names)

(在此之前，我们先假定这些变量都仅在同一个模块内使用。) 这些约定同样也适用于函数命名。对于引用方式设计为 `from M import *` 的模块，应该使用 `__all__` 机制来避免 import 全局变量，或者采用下划线前缀的旧约定来命名全局变量，从而表明这些变量是“模块非公开的"。

函数命名 (Function Names)

函数命名应该都是小写，必要时使用下划线来提高可读性。

只有当已有代码风格已经是混合大小写时（比如 threading.py），为了保留向后兼容性才使用混合大小写。

函数和方法参数 (Function And Method Arguments)

实例方法的第一参数永远都是 `self`。

类方法的第一个参数永远都是 `cls`。

在函数参数名和保留关键字冲突时，相对于使用缩写或拼写简化，使用以下划线结尾的命名一般更好。比如，`class_` 比 `clss` 更好。（或许使用同义词避免这样的冲突是更好的方式。）

方法命名和实例变量 (Method Names And Instance Variables)

使用函数命名的规则：小写单词，必要时使用下划线分开以提高可读性。

仅对于非公开方法和变量命名在开头使用一个下划线。

避免和子类的命名冲突，使用两个下划线开头来触发 Python 的命名修饰机制。

Python 类名的命名修饰规则：如果类 Foo 有一个属性叫 `__a`，不能使用 `Foo.__a` 的方式访问该变量。（有用户可能仍然坚持使用 `Foo._Foo__a` 的方法访问。）一般来说，两个下划线开头的命名方法仅用于避免与设计为子类的类中的属性名冲突。

注意：关于 ___names 的使用也有一些争论（见下）。

常量 (Constants)

常量通常是在模块级别定义的，使用全部大写并用下划线将单词分开。如：`MAX_OVERFLOW` 和 `TOTAL` 。

继承的设计 (Designing For Inheritance)

记得永远区别类的方法和实例变量（属性）应该是公开的还是非公开的。如果有疑虑的话，请选择非公开的；因为之后将非公开属性变为公开属性要容易些。

公开属性是那些你希望和你定义的类无关的客户来使用的，并且确保不会出现向后不兼容的问题。非公开属性是那些不希望被第三方使用的部分，你可以不用保证非公开属性不会变化或被移除。

我们在这里没有使用“私有（private）”这个词，因为在 Python 里没有什么属性是真正私有的（这样设计省略了大量不必要的工作）。

另一类属性属于子类 API 的一部分（在其他语言中经常被称为"protected"）。一些类是为继承设计的，要么扩展要么修改类的部分行为。当设计这样的类时，需要谨慎明确地决定哪些属性是公开的，哪些属于子类 API，哪些真的只会被你的基类调用。

请记住以上几点，下面是 Python 风格的指南：

- 公开属性不应该有开头下划线。
- 如果公开属性的名字和保留关键字有冲突，在你的属性名尾部加上一个下划线。这比采用缩写和简写更好。（然而，和这条规则冲突的是，‘cls’ 对任何变量和参数来说都是一个更好地拼写，因为大家都知道这表示 class，特别是在类方法的第一个参数里。）

  注意 1：对于类方法，参考之前的参数命名建议。

- 对于简单的公共数据属性，最后仅公开属性名字，不要公开复杂的调用或设值方法。请记住，如果你发现一个简单的数据属性需要增加功能行为时，Python 为功能增强提供了一个简单的途径。这种情况下，使用 `Properties` 注解将功能实现隐藏在简单数据属性访问语法之后。

  注意 1：`Properties` 注解仅仅对新风格类有用。

  注意 2：尽量保证功能行为没有副作用，尽管缓存这种副作用看上去并没有什么大问题。

  注意 3: 对计算量大的运算避免试用 properties；属性的注解会让调用者相信访问的运算量是相对较小的。

- 如果你的类将被子类继承的话，你有一些属性并不想让子类访问，考虑将他们命名为两个下划线开头并且结尾处没有下划线。这样会触发 Python 命名修饰算法，类名会被修饰添加到属性名中。这样可以避免属性命名冲突，以免子类会不经意间包含相同的命名。

  注意 1：注意命名修饰仅仅是简单地将类名加入到修饰名中，所以如果子类有相同的类名合属性名，你可能仍然会遇到命名冲突问题。

  注意 2：命名修饰可以有特定用途，比如在调试时，`__getattr__()` 比较不方便。然而命名修饰算法的可以很好地记录，并且容意手动执行。

  注意 3：不是所有人都喜欢命名修饰。需要试着去平衡避免偶然命名冲突的需求和高级调用者使用的潜在可能性。

公开和内部接口 (Public And Internal Interfaces)

任何向后兼容性保证仅对公开接口适用。相应地，用户能够清楚分辨公开接口和内部接口是很重要的。

文档化的接口被认为是公开的，除非文档中明确申明了它们是临时的或者内部接口，不保证向后兼容性。所有文档中未提到的接口应该被认为是内部的。

为了更好审视公开接口和内部接口，模块应该在 `__all` 属性中明确申明公开 API 是哪些。将 `__all__` 设为空 list 表示该模块中没有公开 API。

即使正确设置了 `__all` 属性，内部接口（包，模块，类，函数，属性或其他命名）也应该以一个下划线开头。

如果接口的任一一个命名空间（包，模块或类）是内部的，那么该接口也应该是内部的。

引用的命名应该永远被认为是实现细节。其他模块不应当依赖这些非直接访问的引用命名，除非它们在文档中明确地被写为模块的 API，例如 `os.path` 或者包的 `__init__` 模块，那些从子模块展现的功能。

编程建议 (Programming Recommendations)

- 代码应该以不影响其他 Python 实现 (PyPy, Jython, IronPython, Cython, Psyco 等) 的方式编写。

  例如，不要依赖于 CPython 在字符串拼接时的优化实现，像这种语句形式 `a += b` 和 `a = a + b`。即使是 CPython（仅对某些类型起作用）这种优化也是脆弱的，不是在所有的实现中都不使用引用计数。在库中性能敏感的部分，用`''.join` 形式来代替。这会确保在所有不同的实现中字符串拼接是线性时间的。

- 与单例作比较，像 `None` 应该用 `is` 或 `is not`，从不使用 `==` 操作符。

  同样的, 当心 `if x is not None` 这样的写法, 你是不知真的要判断 `x` 不是 `None`。例如，测试一个默认值为 `None` 的变量或参数是否设置成了其它值，其它值有可能是某种特殊类型（如容器），这种特殊类型在逻辑运算时其值会被当作 `Flase` 来看待。

- 用 `is not` 操作符而不是 `not ... is`。虽然这两个表达式是功能相同的，前一个是更可读的，是首选。

  推荐的写法:

  ```
  if foo is not None:
  ```

  不推荐的写法:

  ```
  if not foo is None:
  ```

- 用富比较实现排序操作的时候，最好实现所有六个比较操作符（`__eq__`、`__ne__`、`__lt__`，`__le__`，`__gt__`，`__ge__`），而不是依靠其他代码来进行特定比较。

  为了最大限度的减少工作量，`functools.total_ordering()` 装饰器提供了一个工具去生成缺少的比较方法。

  PEP 207 说明了 Python 假定的所有反射规则。因此，解释器可能使用 `y > x` 替换 `x < y`，使用 `y >= x` 替换 `x <= y`，也可能交换 `x == y` 和 `x != y` 的操作数。`sort()` 和 `min()` 操作肯定会使用 `<` 操作符，`max()` 函数肯定会使用 `>` 操作符。当然，最好是六个操作符都实现，以便在其他情况下不会出现混淆。

- 始终使用 `def` 语句来代替直接绑定了一个 `lambda` 表达式的赋值语句。

  推荐的写法:

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

  不推荐的写法:

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

  第一个表单意味着生成的函数对象的名称是'f' 而不是通用的'<lambda>'。通常这对异常追踪和字符串表述是更有用的。使用赋值语句消除了使用 `lambda` 表达式可以提供，而一个显式的 `def` 语句不能提供的唯一好处，如，`lambda` 能镶嵌在一个很长的表达式里。

- 异常类应派生自 `Exception` 而不是 `BaseException`。直接继承 `BaseException` 是为 `Exception` 保留的，从 `BaseException` 继承并捕获异常这种做法几乎总是错的。

  设计异常的层次结构，应基于那些可能出现异常的代码，而不是引发异常的位置。编码的时候，以回答“出了什么问题？”为目标，而不是仅仅指出“这里出现了问题”（见 PEP 3151 一个内建异常结构层次的例子）。

  类的命名约定适用于异常，如果异常类是一个错误，你应该给异常类加一个后缀 `Error`。用于非本地流程控制或者其他形式的信号的非错误异常不需要一个特殊的后缀。

- 适当的使用异常链。在 Python 3 里，应该使用 `raise X from Y` 来指示显式替换，而不会丢失原始的追溯。

  当有意替换一个内部的异常时（在 Python 2 用 `raise X`，Python 3.3+ 用 `raise X from None`），请确保将相关详细信息转移到新异常中（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的异常消息中）。

- 在 Python 2 里抛出异常时，用 `raise ValueError('message')` 代替旧式的 `raise ValueError, 'message'`。

  在 Python 3 之后的语法里，旧式的异常抛出方式是非法的。

  使用括号形式的异常意味着，当你传给异常的参数过长或者包含字符串格式化时，你就不需要使用续行符了，这要感谢括号！

- 捕获异常时，尽可能使用明确的异常，而不是用一个空的 except: 语句。

  例如，用：

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

  一个空的 except: 语句将会捕获到 SystemExit 和 KeyboardInterrupt 异常，很难区分程序的中断到底是 Ctrl+C 还是其他问题引起的。

- 如果你想捕获程序的所有错误，使用 `except Exception:`(空 `except:` 等同于 `except BaseException`)。一个好的经验是限制使用空 `except` 语句，除了这两种情况：

  1. 如果异常处理程序会打印出或者记录回溯信息；至少用户意识到错误的存在。
  2. 如果代码需要做一些清理工作，但后面用 `raise` 向上抛出异常。`try .. finally` 是处理这种情况更好的方式。

- 绑定异常给一个名字时，最好使用 Python 2.6 里添加的明确的名字绑定语法：

```
try:
    process_data()
except Exception as exc:
    raise DataProcessingFailedError(str(exc))
```

  Python 3 只支持这种语法，避免与基于逗号的旧式语法产生二义性。

- 捕获操作系统错误时，最好使用 Python 3.3 里引进的明确的异常结构层次，而不是内省的 `errno` 值。

- 另外，对于所有 `try` / `except` 子句，将 `try` 子句限制为必需的绝对最小代码量。同样，这样可以避免屏蔽错误。

  推荐的写法：

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

  不推荐的写法：

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

- 当某个资源仅被特定代码段使用，用 `with` 语句确保其在使用后被立即干净的清除了，`try/finally` 也是也接受的。
- 当它们做一些除了获取和释放资源之外的事的时候，上下文管理器应该通过单独的函数或方法调用。例如：

  推荐的写法：

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

  不推荐的写法：

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

  第二个例子没有提供任何信息来表明 `__enter__` 和 `__exit__` 方法在完成一个事务后做了一些除了关闭连接以外的其它事。在这种情况下明确是很重要的。

- 坚持使用 `return` 语句。函数内的 `return` 语句都应该返回一个表达式，或者 `None`。如果一个 `return` 语句返回一个表达式，另一个没有返回值的应该用 `return None` 清晰的说明，并且在一个函数的结尾应该明确使用一个 `return` 语句（如果有返回值的话）。

推荐的写法：

```
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)
```

不推荐的写法：

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

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

- 用字符串方法代替字符串模块。

  字符串方法总是快得多，并且与 unicode 字符串共享相同的 API。如果需要与 2.0 以下的 Python 的向后兼容，则覆盖此规则。

- 用`''.startswith()`和`''.endswith()`代替字符串切片来检查前缀和后缀。

  `startswith()` 和 `endswith()` 是更简洁的，不容易出错的。例如：

```
# 推荐的写法：
if foo.startswith('bar'):

# 不推荐的写法：
if foo[:3] == 'bar':
```

- 对象类型的比较应该始终使用 `isinstance()` 而不是直接比较。

```
# 推荐的写法：
if isinstance(obj, int):

# 不推荐的写法：
if type(obj) is type(1):
```

当比较一个对象是不是字符串时，记住它有可能也是一个 unicode 字符串！在 Python 2 里面，`str` 和 `unicode` 有一个公共的基类叫 `basestring`，因此你可以这样做：

```
if isinstance(obj, basestring):
```

注意，在 Python 3 里面，`unicode` 和 `basestring` 已经不存在了（只有 `str`），`byte` 对象不再是字符串的一种（被一个整数序列替代）。

- 对于序列（字符串、列表、元组）来说，空的序列为 `False`：

  正确的写法：

  ```
  if not seq:
  if seq:
  ```

  错误的写法：

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

- 不要让字符串对尾随的空格有依赖。这样的尾随空格是视觉上无法区分的，一些编辑器（或者，reindent.py）会将其裁剪掉。

- 不要用 `==` 比较 `True` 和 `False`。

```
# 推荐的写法：
if greeting:

# 不推荐的写法：
if greeting == True:

# 更加不推荐的写法：
if greeting is True:
```

函数注解 (Function Annotations)

随着PEP 484被正式接受，函数注释的样式规则已经改变。

- 为了向前兼容，Python 3 代码中的函数注释最好使用PEP 484语法。（上一节中的有一些关于注解格式的建议。）
- 建议不再使用在此文档早期版本中描述的试验性质的注解样式。
- 然而，在标准库 (stdlib) 之外，现在鼓励在PEP 484的规则范围内的实验。例如，使用PEP 484样式类型的注解标记大型第三方库或应用程序，评估添加这些注解的方式是否简便，并观察其存在是否增加了代码可读性。
- Python 标准库在采用这些注解时应持谨慎态度，但是当编写新代码或进行大的重构时，允许使用。
- 如果希望不按照函数注解的方式来使用函数，可以在文件头部添加以下注释：

  ```
  # type: ignore
  ```

  这会告诉类型检查器忽略所有注解。(在PEP 484中可以找到更细致的方式来控制类型检查器的行为。)

- 像代码扫描工具一样 (linters), 类型检查器是可选的, 单独的工具。默认情况下, Python 解释器不应该由于类型检查而发出任何消息，并且不应该根据注释来改变它们的行为。
- 不想使用类型检查器的用户可以忽略它们。但是，预计第三方库软件包的用户可能希望在这些软件包上运行类型检查器。为此，PEP 484 建议使用存根文件：.pyi 文件，类型检查器优先于相应的.py 文件读取这个文件。存根文件可以与库一起分发，也可以单独地 (在库作者许可的情况下) 通过 Typeshed repo[7]分发。
- 对于需要向后兼容的代码，可以以注释的形式添加类型注解。参见PEP 484 的相关章节。

[7] Typeshed repo https://github.com/python/typeshed

版权 (Copyright)

此文档没有版权限制。

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