# PEP 8: Python代码风格指南

原 文 PEP 8 -- Style Guide for Python Code

作 者
Guido van Rossum (guido@python.org)
Barry Warsaw (barry@python.org)
Nick Coghlan (ncoghlan@gmail.com)

状 态 有效提案

类 型 进程类

发 布 历 史
2001年7月5日
2013年8月1日

## 目录



## 一、导论

- 本文给出的Python代码的编码规范, 适用于主要的Python发行版中组成标准库的Python代码. 请参阅PEP 关于Python的C实现的C编码风格的描述 [1].
- 本文与PEP 257(文档字符串规范)改编自Guido的Python Style Guide一文, 并用Barry的风格指南作了一些补充 [2].
- 本文中的代码风格指南将会随着时间推移而逐渐演变, 伴随着语言自身的变化, 一些过去的约定已经过时, 并且确定了许多新的规定.
- 很多项目都有自己的代码风格, 如若与其发生冲突, 则应优先使用该项目特定的风格.

## 二、A foolish consistency is the hobgoblin of little minds

- Guido的一个重要的观点是读代码的次数总是比写代码要多. 本文给出的风格指南旨在提高代码的可读性并使其在各种Python代码中保持一致性. 正如PEP 20所说的, "可读性非常重要 (Readability counts)".
- 代码风格指南是关于代码编写中的一致性的. 与本文保持一致很重要, 而一个项目中的代码风格一致性更重要, 但一个模块或是功能中的一致性最为重要.
- 可能本文中的建议有时候并不适用, 因此需要知道何时会出现不一致.

  当你出现疑问时, 查找其他例子, 作出最佳判断.

  不要犹豫尽管发问.
- 但是不应为了遵守本篇PEP而破坏代码的向后兼容性!
- 下列情形可以忽略特定的风格规定:

  1. 当应用指南时会降低代码可读性时, 即使对那些习惯于阅读遵守这个PEP的代码的人来说.
  2. 与周围的代码保持一致性也会破坏它(可能处于历史原因).

     虽然这也是个收拾别人烂摊子的好机会(真正的XP风格).
  3. 由于问题代码早于指南的引入, 因此没有修改该段代码的理由.
  4. 当代码需要与不支持风格指南的较早版本Python保持兼容性时.

## 三、代码布局

### 1. 缩进

- 每级缩进使用4个空格.
- 连续行应对齐折叠元素, 无论是用垂直的隐式行连接圆括号, 中括号或大括号内的, 还是使用悬挂缩进. 使用悬挂缩进时应遵循以下几点. 第一行没有参数时应使用更多的缩进用以区别其本身和连续行.

```
# Correct:

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

- 对于连续行来说, 4个空格是规则是可选的.

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

- 当if语句条件部分足够长需要多行时, 值得注意的时两个字符组成的关键字(if)加上一个空格以及一个开扩号为多行条件的后续行创造了一个4个空格的缩进. 这可能会与嵌入if内的缩进语句产生视觉上的冲突. 本篇PEP没有明确如何进一步区分条件行与嵌入行. 在此种情况下, 可接受的方式包括但不限于如下:

```
# 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()
```

- 多行结构中的闭括号可以在最后一行与第一个非空字符对齐.

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

或者也可将其排列在开始多行结构行的第一个字符下.

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

## 2. Tabs还是空格

- 缩进更偏向使用空格.
- tabs应仅在已经使用tabs的代码段中以保持风格一致.
- Python 3不允许在缩进中混用tabs与空格.
- Python 2混用tabs和空格缩进时将会转换为只使用空格.
- 当使用-t选项调用Python命令行解释器时, 对代码混用tabs与空格时会发出警告; 当使用-tt选项时警告将会转变为错误. 强烈建议使用这些选项!

## 3. 行的最大长度

- 每一行最大长度限制在79个字符.
- 对下沉的长块有更少的结构限制(文档字符串或注释), 每行长度应该被限制在72个字符.
- 限制编辑器窗口宽度可以并排打开多个文件, 并且使用代码审查工具显示相邻列的两个版本时效果很好.
- 大多数工具的默认折叠会破坏代码的视觉结构, 使其更难理解. 将编辑器的窗口宽度设置为80个字符以避免折叠代码, 即使其在最后一行放置一个标记字形. 而一些基于网络的工具很可能不提供动态换行.
- 某些团队更偏向于使用较长的行长度. 完全或主要由一个团队维护代码的, 可以在这个问题上达成一致的, 可以将行的最大长度增加到99个字符, 但注释和文档字符串的长度仍然限制在72个字符.

Python标准库选择了保守的做法, 要求行限制到79个字符(注释/文档字符串限制在72个字符).

- 折叠长行的首选方式是在小括号, 中括号, 大括号中使用Python隐式换行. 长行可以在表达式外使用小括号分成多行. 应该优先使用此种方式而非使用反斜杠续行.

- 反斜杠有时仍是合适的. 例如长的多行with语句不能使用隐式续行, 因此可以使用反斜杠.

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

另外一个类似的例子是assert语句.

- 确保适当的连续行缩进.

## 4. 换行应在二元运算符前还是其后

- 数十年来被推荐的代码风格是在二元运算符后换行, 但是这种方式会破坏代码的可读性: 运算符被分散在屏幕的不同列上, 此外运算符与被运算对象不再同一列上, 分辨进行了何种运算较为麻烦.

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

- 为了解决可读性的问题, 数学家和印刷业者通常在二元运算符前换行. Donald Knuth在其Computers and Typesetting系列文章中解释了这个规则:

> "虽然写在一段话中的公式经常在二元运算符后换行, 但在单独展示的公式中通常是在二元运算符前换行." [3]

- 遵循数学传统则代码将更具可读性.

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

- 在现有文件中, 在二元运算符之前或其后换行应与原有代码保持一致, 但在新的代码中, 建议使用Knuth风格即在二元运算符前换行.

## 5. 空行

- 在顶级函数和类的定义之间应有两行空行.
- 类中方法的定义之间应有一行空行.
- 分割相关的函数组谨慎地使用额外空行. 一组相关的行之间不使用空行.
- 在函数中谨慎使用空行表示逻辑部分.
- Python允许使用control-L换页符作为空白符; 许多工具将其视作分页符, 所以也可以使用其为文件中的相关部分分页. 需要注意的是, 一些编辑器和基于Web的代码查看器可能不能将control-L识别为换页符, 而是显示为另外的字形.

## 6. 源文件编码

- 代码应始终使用UTF-8编码(或在Python 2中使用ASCII编码).
- 使用ASCII编码(Python 2)或UTF-8编码(Python 3)的文件无须编码声明.
- 在标准库中, 非默认编码应仅用于测试或当注释或文档字符串需要提及的作者名包含非ASCII字符; 否则, 使用\x, \u, \U, \N是在字符串中包含非ASCII数据的首要方式.
- 对于Python 3.x, 为标准库规定以下标准(参见PEP 3131): Pyhton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**使用英文单词(在许多情况下使用的缩写和技术术语并非英语). 此外, 字符串和注释也必须使用ASCII. 仅有的列外是:

    1. 测试非ASCII的特性;
    2. 作者名.

    作者名不是基于拉丁字母(latin-1, ISO/IEC 8859-1字符集)的**必须**提供其名字拉丁字母的音译.
- 鼓励面向的开源项目采用类似的策略.

## 7. 库的导入

- 库的导入通常在不同行.

```
# Correct
import os
import sys

# Wrong
import os, sys
```

- 以下方式是允许的:

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

- 库的导入应在文档的顶端, 在模块注释和文档字符串之后, 而在模块全局变量和常量之前.

  导入应当遵循以下顺序分组:

  1. 标准库;
  2. 相关的第三方库
  3. 特定的本地应用/库

  应当在每组导入之间加入一个空行.

- 推荐使用绝对引用, 因为其更具可读性且若导入系统配置错误时有更好的表现.

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

  但清晰的相对导入是绝对导入的一种可接受的替代方式, 特别是在处理复杂的包布局时, 使用绝对导入将会过于冗长.

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

  标准库代码应该避免使用复杂的包布局并使用绝对导入.

  绝对不要使用隐式的相对导入, 并且在Python 3中已经将其删除.

- 当从一个包含类的模块中导入类时:

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

  如果会与本地命名空间产生冲突, 则:

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

  并使用"myclass.MyClass"和"foo.bar.yourclass.YourClass"访问.

- 应当避免使用通配符导入:

```
from module import *
```

通配符会使得当前命名空间中存在的命名不清晰. 但当将内部接口作为一个公共API的一部分时(如重写一个纯Python实现的接口, 该接口定义从一个可选的加速器模块并且哪些定义将被重写并不提前知晓), 此时是使用通配符的合理用例.

- 使用这种方式重新命名时, 后文关于公共和内部接口的准则仍然适用.

### 8. 模块级别的内置属性

- 模块级别的内置属性(名字前后有双下划线的), 例如__all__, **author**, __version__等, 应放置在模块的文档字符串后, 任意import语句之前, from __future__导入除外. Python强制要求from __future__导入必须在任何代码之前, 只能在模块级文档字符串之后.

```
"""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
```

## 四、字符串引号

- 在Python中, 单引号字符串与双引号字符串是相同的. 本篇PEP并不建议如此. 选取一个规范并一直使用同一风格. 但字符串中包含单引号字符或双引号字符时, 使用与你代码风格相反的另一种引号以避免使用反斜杠. 这样可以提升代码的可读性

- 对于三引号字符串, 建议总是使用双引号字符, 与PEP 257保持一致.

## 五、表达式和语句中的空格

### 1. Pet Peeves 一点点讨厌的小毛病

- 在以下情况避免使用额外的空格:

- 紧挨着小括号, 中括号或大括号的.

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

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

- 在尾随的逗号和其后右括号之间的.

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

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

- 紧挨在逗号, 分号或冒号前的.

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

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

- 切片中的冒号与二元运算符类似, 应在其两侧都有相同数量的空格(可将其视为优先级最低的运算符). 在拓展切片中, 两个冒号必须使用相同数量的空格. 但当一个切片参数被省略时, 该空格也被省略.

```
# 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 : : upper]
ham[ : upper]
```

- 紧挨在左括号之前的, 在函数调用的参数列表开始处.

```
# Correct:
spam(1)

# Wrong:
spam (1)
```

- 紧挨着索引或切片开始的左括号之前的.

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

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

- 在赋值或其他运算符周围为了与其他行对齐而多于一个空格.

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

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

## 2. 其他建议

- 始终避免尾行空白. 尾行空白通常是不可见的, 容易造成困惑: 如在反斜杠后跟一空格, 则其就不是一个有效的续行符了. 一些编辑器不保留尾行空白, 许多项目如Cpython也设置了在commit前检查以拒绝尾行空白的存在.

- 总是在二元运算符两侧放置一个空格: 赋值(=), 增强赋值(+=, -=, 等), 比较(==, <, >, !=, <=, >=, in, not, is, is not), 布尔运算符(and, or, not).

- 若使用了不同优先级的运算符, 在优先级较低的运算符两侧增加空格. 但决不能超过一个空格, 且在二元操作符两侧空格数量相同.

```
# 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)
```

- 函数注解应当使用正常的冒号规则, 并应在->箭头两端留有空格.

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

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

- 当用于表明关键字参数或用于未注释的函数参数的默认值时, 在等号周围无须空格.

```
# 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)
```

若参数即有注解又有默认值时, 在等号两侧各有一空格.

```
# 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): ...
```

- 复合语句(同一行有多个语句)不建议使用.

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

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

- 尽管有时可以在if/for/while后同一行跟一小段代码, 但绝对不要在多个子语句中这样使用. 避免折叠长行.

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

# DEFINITELY NOT!
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()
```

## 六、何时使用尾部逗号

- 尾部逗号通常是可选的, 除了一些强制的情况, 如元组只有一个元素时. 为了使代码更清晰, 元组只有一个元素时建议使用括号括起来.

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

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

- 当尾部逗号不是必须的, 如若使用了版本控制系统时那么其将非常有用. 当列表元素, 参数, 导入项未来可能不断增加时, 留一个尾部逗号是一个很好的选择. 通常的用法是每个元素独占一行, 然后尾部都有逗号, 在最后一个元素的下一行写闭标签. 如若数据结构都是写在同一行的, 就没有必要保留尾部逗号了.

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

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

# 七、注释

- 与代码相矛盾的注释比没有注释还要糟糕. 当修改代码时一定要及时更新注释.
- 注释应当为完整的句子. 第一个单词的首字母应当大写, 除非其为由小写字母开头的标识符.
- 注释块通常包含一段或多段由完整的句子构成的段落, 每个句子都应以句号结尾.
- 在多行注释中, 除了最后一句外, 应在句尾的句号后增加两个空格.
- 确保你的注释清晰明了, 并对其他使用该语言的人来说也是如此.
- 对于非英语国家的Python程序员, 请使用英语编写注释, 除非你120%确信你的代码不会被与你使用不同语言的人读到.

## 1. 注释块

- 注释块通常适用于紧跟其后的一些(或全部)代码, 并且与这些代码有着相同级别的缩进. 注释块的每行以一个#和一个空格开始(除非注释文本有缩进).
- 注释块内的段落之间由仅包含#的行隔开.

## 2. 行内注释

- 谨慎使用行内注释.
- 行内注释与代码在同一行, 其与代码之间应至少相隔两个空格, 并以#和一个空格开头.
- 若代码所表明的意义明显则行内注释是不必要的.

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

但有时是有用的:

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

## 3. 文档字符串

- 关于写出好的文档字符串的约定在PEP 257中.
- 为所有公共模块, 函数, 类和方法编写文档字符串. 对私有方法编写文档字符串是不必要的, 但应当在def语句的下一行为该方法编写描述性的注释.
- PEP 257详细描述了好的文档字符串的规定. 需要注意到最重要的是, 文档字符串结束的三引号"""应单独一行.

```
"""Return a foobang

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

- 对单行文档字符串, 应将"""写在同一行.

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

## 八、命名规范

- Python库的命名规范有些微混乱, 所以我们不会使其变得完全一致. 但是, 这是目前推荐的命名标准. 新模块和包(包括第三方框架)应该按照这些标准编写, 但对现有的有不同风格的库, 保持内部一致性是首选的.

### 1. 首要原则

- 用户可见的API的公开部分的名称, 应该遵循反映用法而非实现的约定.

### 2. 描述: 命名风格

- 有很多不同的命名风格. 独立于其用途, 有助于识别使用了何种命名风格.
- 以下的命名风格是最常见的:
    - b (单个小写字母)
    - B (单个大写字母)
    - lowercase (小写字符串)
    - lower_case_with_underscores (带下划线的小写字符串)
    - UPPERCASE (大写字符串)
    - UPPER_CASE_WITH_UNDERSCORES (带下划线的大写字符串)
    - CapitalizedWords (单词首字母大写字符串, 或CapWords, 或驼峰命名法 [4])

      需要注意的是, 当在CapWords中使用缩写时, 大写所有的缩写字母. 因此HTTPServerError优于HttpServerError.
    - mixedCase (混用大小写的字符串, 与单词首字母大写字符串不同的是其第一个单词为小写)
    - Capitalized_Words_With_Underscores (极其丑陋!)

- 还有一种使用简短独特的前缀将相关名字组织在一起的风格. 这在Python中用的不多, 但是为了完整起见在此提及. 例如, os.stat()函数返回一个元组, 其元素名字通常为st_mode, st_size, st_mtime等类似样式. (这样做是为了强调与POSIX系统调用的字段的对应关系, 这有助于程序员熟悉该结构.)
- X11库的所有公开函数以X开头. 在Python中这种方式通常认为是不必要的, 因为属性名和方法名以对象名作为前缀, 而函数名以模块名为前缀.
- 此外, 使用前导下划线或后置下划线的特殊形式也是可以识别的(通常可以将其与任意规定相结合使用):
    - _single_leading_underscore (单前导下划线): 弱"内部使用"标志. 如from M import * 不会导入以下划线开头的对象.
    - single_trailing_underscore_ (单后置下划线): 约定用于避免与Python关键字冲突. 如:

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

    - __double_leading_underscore (双前导下划线): 命名类属性, 调用时名称混淆/名称修饰(name mangle)(在FooBar类中, __boo将变成_FooBar__boo, 详见下文).
    - __double_leading_and_trailing_underscore__ (双前导与后置下划线): 存在于用户控制的命名空间中的Magic对象或属性. 如__init__, __import__或__file__. 不要创造这样的名称. 仅像文档中一样使用它们.

## 3. 规定: 命名规范

**避免使用的名字**

- 绝对不要使用字符"l", "O"以及"I"作为单字符变量名.

  在某些字体中, 这些字符与数字0和1不容易区分. 当想使用"l"时, 用"L"代替.

**ASCII兼容性**

- 如PEP 3131所述, 标准库中使用的标识符必须与ASCII兼容.

**包与模块名**

- 模块名应简短且全为小写字符. 如若可提升可读性可以使用下划线. Python包名也应简短且全为小写字符, 但是不鼓励使用下划线.
- 当一个由C或C++编写的拓展模块, 伴随Python模块可以实现更高水平(如更面向对象)的接口时, C/C++模块名前应有一个下划线(如_socket).

**类名**

- 类名通常使用CapWords规范.
- 当接口被记录且主要用作可调用时, 函数命名规则可被替换.

- 需要注意到的是, 内置名有一个单独的规则: 大多数内置名是单个单词(或两个单词一起), CapWords规则仅用于异常名和内置常量.

**类型变量名称**

- 由PEP 484引入的类型变量名使用CapWords规则, 且偏向于使用较短的名字: T, AnyStr, Num等. 建议在用于声明协变或反变行为的变量之后添加_co或_contra后缀.

```
from typing import TypeVar

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

**异常名**

- 由于异常为类, 因此类的命名规范可在此适用. 但你应该在异常名后加上Error后缀.

**全局变量名**

- (但愿这些变量只会在一个模块中适用)全局变量的命名规则与函数的命名规则相同.
- 模块设计为通过from M import * 来调用, 应使用__all__机制防止导出全局变量, 或使用加前缀的旧规则, 为全局变量加下划线(你可能想表明这些全局变量为非公开模块).

**函数和变量名**

- 函数和变量名应为小写字符, 尽量使用下划线将不同单词分开以提升可读性.
- 变量名与函数名遵守相同的命名规则.
- 为了保持向后兼容性, 混用大小写的规则仅在其以成为主要风格的代码中使用(如threading.py)

**函数与方法名**

- 使用self作为实例化方法的第一个参数.
- 使用cls作为类方法的第一个参数.
- 如若函数参数名与保留关键字发生冲突, 最好在末尾附加一个下划线, 而不要使用缩写或错误拼写. 因此, class_优于clss. (也许更好的办法是使用同义词避免此类冲突.)

**方法名与实例变量**

- 使用函数命名规则: 小写字符, 必要时可使用下划线将单词分开以提高可读性.

  仅在非公共方法和实例变量前使用前导下划线.

- 为了避免名称与子类冲突, 使用两个前导下划线以调用Python的名称混淆规则.

Python用类名修饰这些名字: 如果Foo类有一个名为__a的属性, 则Foo.__a不能访问它. (可以通过Foo._Foo__a访问.) 通常应仅使用双前导下划线来避免与子类属性名冲突.

注意: 关于__name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, 使用下划线分隔单词且所有字母大写. 如MAX_OVERFLOW, TOTAL.

**继承的设计**

- Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.
- Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.
- We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).
- Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.
- With this in mind, here are the Pythonic guidelines:
    - Public attributes should have no leading underscores.
    - If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

      Note 1: See the argument name recommendation above for class methods.
    - For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

      Note 1: Properties only work on new-style classes.

      Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

- If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

  Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

  Note 2: Name mangling can make certain uses, such as debugging and __getattr__(), less convenient. However the name mangling algorithm is well documented and easy to perform manually.

  Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

4. 公共和内部接口

- 任何向后兼容性保证只适用于公共接口. 因此重要的是用户能够清除的区分公共接口和内部接口.
- 文档接口除非文档明确声明其为临时接口或内部接口, 不受通常的向后兼容性保证, 否则文档接口被视为公共接口. 所有非文档化的接口都被假定为内部接口.
- 为了更好的支持自省, 模块应使用__all__属性显式声明其公开的API的名字, __all__设置为一个空列表时表示该模块没有公开API.

  即使适当地设置了__all__, 内部接口(包, 模块, 类, 函数, 属性或其他名字)仍应以一个前导下划线作为前缀.
- 如若一个接口包含命名空间(包, 模块或类), 则该接口被认为是内部接口.
- 导入的名称应始终被认为是实现细节. 其他模块不得以来对此类导入名的间接访问, 除非其是一个明确的记录包含模块的API的一部分. 如os.path或从子模块公开的包的__init__模块.

# 九、编程建议

- 编写的代码应不损害其他方式的Python实现(PyPy, Jython, IronPython, Cython, Psyco等).

  如, 对于形如+=b或a=a+b的语句, 不要依赖于Cython有效的实现就地字符串连接. 即使在Cython中, 这种优化也是脆弱的(仅适用于某些类型), 并且在不使用引用计数的实现中不存在这种优化. 在库的性能易受影响的部分, 应该使用.join()形式. 这样可以保证在各种实现方式中连接可以在线性时间中完成.
- 与单值如None比较应使用is, is not, 不要使用等号操作符.

  同样的, 如果你想表达的真实意思是if x is not None, 谨防写成if x. 如, 当测试一个默认为None的变量或参数是否被赋为其他值时. 这个其他值可能具有在布尔上下文中为false的类型(如容器).
- 使用is not操作符而非not...is. 尽管这两个表达式功能相同, 但前者更具可读性且更优.

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

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

- 当执行具有丰富比较的排序操作时, 最好执行所有的六个操作符(__eq__, __ne__, __lt__, __le__, __gt__, __ge__)而非依赖其他代码只能进行一个特定的比较.

  为了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函数.

  PEP 207表明, Python假定自反性规则. 因此编译器可以交换y > x和x < y, y >= x和x <= y, 也可以交换参数x == y和x != y. sort()和min()操作保证使用<操作符而max()函数使用>操作符. 不管怎样, 最好实现六个操作, 这样其在上下文就不会产生混淆.

- 使用def语句而非赋值语句将lambda表达式绑定到标识符上.

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

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

  第一种形式意味着所得的函数对象名称为"f"而非一般的"". 这对于回溯和字符串标识更有用. 使用赋值语句消除了lambda表达式相对于显式def语句可以提供的唯一好处(即可以将其嵌入较大的表达式中).

- 从Exception而非BaseException中派生出异常. 直接继承BaseException是保留那些捕捉几乎总是错的异常的.

  根据可能需要捕获异常的代码(而非引发异常的位置)的区别设计异常的层次结构. 旨在以编程方式回答"出了什么问题?", 而非仅仅说明"出现了问题"(参阅PEP 3151以了解针对内置异常结构的一个例子).

  类的命名规则在此适用, 但如果异常是错误, 则应在异常类后添加"Error"后缀. 用于非本地控制流或其他形式的信号的非错误异常, 无需添加后缀.

- 适当的适用异常链. 在Python 3中, "raise X from Y"应用于表明明确的替换而不失去原先追踪到的信息.

  当故意替换一个内部异常时(Python 2中使用"raise X"而Python 3.3+中使用"raise X from None"), 确保相关细节被转移到新的异常中(比如当转换KeyError为AttributeError时保留属性名, 或在新的异常消息中嵌入原始异常的文本).

- 当在Python 2中引发异常时, 请使用raise ValueError("message")而非较老的形式raise ValueError, "Message".

  后一种形式不是合法的Python 3语法.

  目前使用的形式意味着当异常的参数很长或包含格式化字符串时, 多亏了小括号不必再使用续行符.

- 当捕获到异常时, 尽可能提及特定的异常, 而非使用空的"except:"子句.

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

空的"except:"子句将捕获SystemExit和KeyboardInterrupt异常, 这使得很难用Control-C来中断程序, 也会掩饰其他的问题. 如果想捕获会导致程序错误的所有异常, 使用"except Exception:"(空异常相当于"except BaseException:").

一个好的经验法则是将空"except"子句的使用限制为两种情况:

1. 如果异常处理程序将打印输出或记录回溯; 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, 但随后让异常用raise抛出. 处理这种情况用try...finally更好.

- 在将捕获的异常绑定到一个名称时, 最好是用Python 2.6中添加的显式名称绑定语法:

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

这是Python 3中唯一支持的语法, 并且避免与旧的基于逗号的语法相关的歧义问题.

- 捕获操作系统异常时, 最好使用Python 3.3中引进的明确的异常层次结构, 而非通过errno值的自省.

- 此外, 对于所有try/except子句, 请将try子句限制为所需要的绝对最s少代码量. 这避免了掩盖错误:

```
# 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)
```

- 当一个资源是一个局部特定的代码段时, 使用with语句以确保在使用后立即可靠地对其清理. 也可使用try/finally语句.

- 无论何时做了除获取或释放资源的的一些操作, 应通过单独的函数或方法调用上下文管理器:

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

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

  后一个例子没有提供任何信息来表明__enter__和__exit__方法除了在事务结束后关闭连接外, 还在做其他事情. 在这种情况下, 明确很重要.

- 返回语句应保持一致性. 函数中的所有return语句都有返回值, 或都没有返回值. 如若任何return语句有返回值, 那么没有返回值的任何return语句应将其显式声明为return None, 并且在函数末尾(如果可访问)应存在一个显式的return语句:

```
# 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)
```

- 使用字符串方法而非string模块.

  字符串方法总是更快且与unicode字符串使用相同的API. 如果必须向后兼容Python 2.0以前的版本, 可以无视这个原则.

- 使用".startswith()"和".endswith()"代替字符串切片来检查前缀和后缀.

startswith()和endswith()更清晰, 并错误率很少.

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

# Wrong:
if foo[:3] == "bar":
```

- 对象类型的比较应使用isinstance()替代直接比较:

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

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

当检查对象是否为字符串时, 请注意其也可能是一个unicode字符串! Python 2中, str和unicode有共同的基类basestring, 所以你可以这么做:

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

注意在Python 3中, unicode和basestring不再存在(只有str)且字节对象不再是字符串(而是integers序列).

- 对于序列(字符串, 列表, 元组), 可利用空序列为false:

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

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

- 不要编写依赖后置空格的字符串. 这些后置空格在视觉上无法区分, 并且有些编辑器将会去除它们.

- 不要用==将布尔值与True或False进行比较:

```
# Correct:
if greeting:

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

```
# Worse:
if greeting is True:
```

- 不鼓励在try...finally中使用流控制语句return/break/continue, 在该方法中, 流控制语句将跳到finally套件之外. 这是因为这样的语句将会隐性的取消通过finally套件传播的任何异常活动:

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

## 1. 函数注解

- 接受PEP 484后, 函数注解的样式规则正在改变.
- 为了向前兼容性, Python 3中的函数注解最好应使用PEP 484语法.
- 不再鼓励使用在本PEP前建议的注解样式进行实验.
- 但在标准库之外, 现在是鼓励在PEP 484规则子类进行实验. 如使用PEP 484风格的类型注解为大型第三方库或程序标记, 查看这些注解的难易程度, 并观察其存在是否增加了代码的可理解性.
- Python标准库在采用此类注解时应保持保守, 但允许将其用于新代码和大型重构中.
- 对于向将函数注解做不同用处的代码, 推荐添加以下形式的注释:

```
# type: ignore
```

  写在文件顶部. 这告诉类型检查器忽视所有注解.

- 像检查器一样, 类型检查器是可选且独立的工具. 默认的Python解释器不应由类型检查而发出信息, 也不应基于注解而改变行为.
- 不愿使用类型检查器的用户可以自由忽略类型检查器. 然而, 第三方库的用户应对那些软件包运行类型检查器. 为此, PEP 484建议使用存根文件: 类型检查其优先于相应的.py文件读取的.pyi文件. 存根文件可以与库一起分发, 也可以通过排版的仓库单独分发(在库作者的允许下). [5])
- 对于需要保持向后兼容性的代码, 类型注解可以注释的形式添加. 详见PEP 484的相关部分. [6])

## 2. 变量注解

- PEP 526引入了变量注解. 对于其的风格建议与上文提到的对函数注解的建议类似.
- 对于模块级变量, 类和实例变量以及局部变量的注解, 在冒号后应有一空格.

- 在冒号前不应有空格.
- 赋值应在等号两边都有一个空格.

```
# 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PEP 526被Python 3.6接受, 但变量注解语法是所有版本的Python存根文件中的首选语法(详见PEP 484).

## 十、参考文献

[1] PEP 7, Style Guide for C Code, van Rossum

[2] Barry's GNU Mailman Style Guide http://barry.warsaw.us/software/STYLEGUIDE.txt
[3] Donald Knuth's The TeXBook, pages 195 and 196.
[4] http://www.wikipedia.com/wiki/CamelCase
[5] Typeshed Repo https://github.com/python/typeshed
[6] Suggested Syntax for Python 2.7 and Straddling Code https://www.python.org/dev/peps/pep-0484/#suggested-syntax-for-python-2-7-and-straddling-code