Python PEP8 编码规范中文版

Introduction 介绍

本文提供的 Python 代码编码规范基于 Python 主要发行版本的标准库。Python 的 C 语言实现的 C 代码规范请查看相应的 PEP 指南 [1]。

这篇文档以及 PEP 257（文档字符串的规范）改编自 Guido 原始的《Python Style Guide》一文，同时添加了一些来自 Barry 的风格指南 [2]。

这篇规范指南随着时间的推移而逐渐演变，随着语言本身的变化，过去的约定也被淘汰了。

许多项目有自己的编码规范，在出现规范冲突时，项目自身的规范优先。

A Foolish Consistency is the Hobgoblin of Little Minds 尽信书,则不如无书

Guido 的一条重要的见解是代码阅读比写更加频繁。这里提供的指导原则主要用于提升代码的可读性，使得在大量的 Python 代码中保持一致。就像 PEP 20 提到的，“Readability counts”。

这是一份关于一致性的风格指南。这份风格指南的风格一致性是非常重要的。更重要的是项目的风格一致性。在一个模块或函数的风格一致性是最重要的。

然而，应该知道什么时候应该不一致，有时候编码规范的建议并不适用。当存在模棱两可的情况时，使用自己的判断。看看其他的示例再决定哪一种是最好的，不要羞于发问。

特别是不要为了遵守 PEP 约定而破坏兼容性！

几个很好的理由去忽略特定的规则：

1. 当遵循这份指南之后代码的可读性变差，甚至是遵循 PEP 规范的人也觉得可读性差。
2. 与周围的代码保持一致（也可能出于历史原因），尽管这也是清理他人混乱（真正的 Xtreme Programming 风格）的一个机会。
3. 有问题的代码出现在发现编码规范之前，而且也没有充足的理由去修改他们。
4. 当代码需要兼容不支持编码规范建议的老版本 Python。

Code lay-out 代码布局

Indentation 缩进

每一级缩进使用 4 个空格。

续行应该与其包裹元素对齐，要么使用圆括号、方括号和花括号内的隐式行连接来垂直对齐，要么使用挂行缩进对齐 3。当使用挂行缩进时，应该考虑到第一行不应该有参数，以及使用缩进以区分自己是续行。

推荐：

```
# 与左括号对齐
foo = long_function_name(var_one, var_two,
                      var_three, var_four)

# 用更多的缩进来与其他行区分
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 挂行缩进应该再换一行
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

不推荐：

```
# 没有使用垂直对齐时，禁止把参数放在第一行
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 当缩进没有与其他行区分时，要增加缩进
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

四空格的规则对于续行是可选的。

可选：

```
# 挂行缩进不一定要用 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换行写的时候，注意可以在两个字符关键字的连接处（比如 if），增加一个空格，再增加一个左括号来创造一个 4 空格缩进的多行条件。这会与 if 语句内同样使用 4 空格缩进的代码产生视觉冲突。PEP 没有明确指明要如何区分 i 发的条件代码和内嵌代码。可使用的选项包括但不限于下面几种情况：

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

# 增加一个注释，在能提供语法高亮的编辑器中可以有一些区分
if (this_is_one_thing and
    that_is_another_thing):
    # Since both conditions are true, we can frobnicate.
    do_something()

# 在条件判断的语句添加额外的缩进
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

（可以参考下面关于是否在二进制运算符之前或之后截断的讨论）

在多行结构中的大括号/中括号/小括号的右括号可以与内容对齐单独起一行作为最后一行的第一个字符，就像这样：

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

或者也可以与多行结构的第一行第一个字符对齐，就像这样：

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

Tabs or Spaces?　制表符还是空格?

空格是首选的缩进方式。

制表符只能用于与同样使用制表符缩进的代码保持一致。

Python3 不允许同时使用空格和制表符的缩进。

混合使用制表符和空格缩进的 Python2 代码应该统一转成空格。

当在命令行加入-t 选项执行 Python2 时，它会发出关于非法混用制表符与空格的警告。当使用–tt 时，这些警告会变成错误。强烈建议使用这样的参数。

Maximum Line Length 行的最大长度

所有行限制的最大字符数为 79。

没有结构化限制的大块文本（文档字符或者注释），每行的最大字符数限制在 72。

限制编辑器窗口宽度可以使多个文件并行打开，并且在使用代码检查工具(在相邻列中显示这两个版本)时工作得很好。

大多数工具中的默认封装破坏了代码的可视化结构，使代码更难以理解。避免使用编辑器中默认配置的 80 窗口宽度，即使工具在帮你折行时在最后一列放了一个标记符。某些基于 Web 的工具可能根本不提供动态折行。

一些团队更喜欢较长的行宽。如果代码主要由一个团队维护，那这个问题就能达成一致，可以把行长度从 80 增加到 100 个字符（更有效的做法是将行最大长度增加到 99 个字符），前提是注释和文档字符串依然已 72 字符折行。

Python 标准库比较保守，需要将行宽限制在 79 个字符（文档/注释限制在 72）。

较长的代码行选择 Python 在小括号，中括号以及大括号中的隐式续行方式。通过小括号内表达式的换行方式将长串折成多行。这种方式应该优先使用，而不是使用反斜杠续行。

反斜杠有时依然很有用。比如，比较长的，多个 with 状态语句，不能使用隐式续行，所以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())
```

（请参阅前面关于多行 if-语句的讨论，以获得关于这种多行 with-语句缩进的进一步想法。）

另一种类似情况是使用 assert 语句。

确保在续行进行适当的缩进。

Should a line break before or after a binary operator? 在二元运算符之前应该换行吗?

几十年来，推荐的风格是在二元运算符之后中断。但是这回影响可读性，原因有二：操作符一般分布在屏幕上不同的列中，而且每个运算符被移到了操作数的上一行。下面例子这个情况就需要额外注意，那些变量是相加的，那些变量是相减的：

```
# 不推荐: 操作符离操作数太远
income = (gross_wages +
         taxable_interest +
         (dividends - qualified_dividends) -
         ira_deduction -
         student_loan_interest)
```

为了解决这种可读性的问题，数学家和他们的出版商遵循了相反的约定。Donald Knuth 在他的 Computers and Typesetting 系列中解释了传统规则： “尽管段落中的公式总是在二元运算符和关系之后中断，显示出来的公式总是要在二元运算符之前中断”[4]。

遵循数学的传统能产出更多可读性高的代码：

```
# 推荐：运算符和操作数很容易进行匹配
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，只要本地的约定是一致的。

对于新代码，建议使用 Knuth 的样式。

Blank Lines 空行

顶层函数和类的定义，前后用两个空行隔开。

类里的方法定义用一个空行隔开。

相关的功能组可以用额外的空行（谨慎使用）隔开。一堆相关的单行代码之间的空白行可以省略（例如，一组虚拟实现 dummy implementations）。

在函数中使用空行来区分逻辑段（谨慎使用）。

Python 接受 control-L（即^L）换页符作为空格；许多工具把这些字符当作页面分隔符，所以你可以在文件中使用它们来分隔相关段落。请注意，一些编辑器和基于 Web 的代码阅读器可能无法识别 control-L 为换页，将在其位置显示另一个字形。

Source File Encoding 源文件编码

Python 核心发布版本中的代码总是以 UTF-8 格式编码（或者在 Python2 中用 ASCII 编码）。

使用 ASCII（在 Python2 中）或 UTF-8（在 Python3 中）编码的文件不应具有编码声明。

在标准库中，非默认的编码应该只用于测试，或者当一个注释或者文档字符串需要提及一个包含内 ASCII 字符编码的作者名字的时候；否则，使用\x,\u,\U，或者 \N 进行转义来包含非 ASCII 字符。

对于 Python 3 和更高版本，标准库规定了以下策略（参见 PEP 3131）：Python 标准库中的所有标识符必须使用 ASCII 标识符，并在可行的情况下使用英语单词（在许

多情况下，缩写和技术术语是非英语的）。此外，字符串文字和注释也必须是ASCII。唯一的例外是（a）测试非 ASCII 特征的测试用例，以及（b）作者的名称。作者的名字如果不使用拉丁字母拼写，必须提供一个拉丁字母的音译。

鼓励具有全球受众的开放源码项目采取类似的政策。

Imports 导入

- 导入通常在分开的行，例如：

```
推荐: import os
    import sys

不推荐:  import sys, os
```

但是可以这样：

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

- 导入总是位于文件的顶部，在模块注释和文档字符串之后，在模块的全局变量与常量之前。

  导入应该按照以下顺序分组：

  1. 标准库导入
  2. 相关第三方库导入
  3. 本地应用/库特定导入

  你应该在每一组导入之间加入空行。

- 推荐使用绝对路径导入，如果导入系统没有正确的配置（比如包里的一个目录在sys.path 里的路径后），使用绝对路径会更加可读并且性能更好（至少能提供更好的错误信息）：

```
import mypkg.sibling

from mypkg import sibling

from mypkg.sibling import example
```

然而，显示的指定相对导入路径是使用绝对路径的一个可接受的替代方案，特别是在处理使用绝对路径导入不必要冗长的复杂包布局时：

```
from . import sibling

from .sibling import example
```

标准库要避免使用复杂的包引入结构，而总是使用绝对路径。

不应该使用隐式相对路径导入，并且在 Python 3 中删除了它。

- 当从一个包含类的模块中导入类时，常常这么写：

```
from myclass import MyClass

from foo.bar.yourclass import YourClass
```

如果上述的写法导致名字的冲突，那么这么写：

```
import myclass

import foo.bar.yourclass
```

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

- 避免通配符的导入（from import *），因为这样做会不知道命名空间中存在哪些名字，会使得读取接口和许多自动化工具之间产生混淆。对于通配符的导入，有一个防御性的做法，即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，以及重写那些事先不知道的定

义）。

当以这种方式重新发布名称时，以下关于公共和内部接口的准则仍然适用。

Module level dunder names 模块级的“呆”名

像`__all__`，`__author__`，`__version__`等这样的模块级“呆名“（也就是名字里有两个前缀下划线和两个后缀下划线），应该放在文档字符串的后面，以及除from `__future__`之外的 import 表达式前面。Python 要求将来在模块中的导入，必须出现在除文档字符串之外的其他代码之前。

比如：

```
"""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
```

String Quotes 字符串引号

在 Python 中，单引号和双引号字符串是相同的。PEP 不会为这个给出建议。选择一条规则并坚持使用下去。当一个字符串中包含单引号或者双引号字符的时候，使用和最外层不同的符号来避免使用反斜杠，从而提高可读性。

对于三引号字符串，总是使用双引号字符来与 PEP 257 中的文档字符串约定保持一致。

Whitespace in Expressions and Statements 表达式和语句中的空格

Pet Peeves 不能忍受的事情

在下列情况下，避免使用无关的空格：

- 紧跟在小括号，中括号或者大括号后。

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

No:  spam( ham[ 1 ], { eggs: 2 } )
```

- 紧贴在逗号、分号或者冒号之前。

```
Yes: if x == 4: print x, y; x, y = y, x

No:  if x == 4 : print x , y ; x , y = y , x
```

- 然而，冒号在切片中就像二元运算符，在两边应该有相同数量的空格（把它当做优先级最低的操作符）。在扩展的切片操作中，所有的冒号必须有相同的间距。

  例外情况：当一个切片参数被省略时，空格就被省略了。

  推荐：

```
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]
```

不推荐：

```
ham[lower + offset:upper + offset]

ham[1: 9], ham[1 :9], ham[1:9 :3]

ham[lower : : upper]

ham[ : upper]
```

- 紧贴在函数参数的左括号之前。

```
Yes：spam(1)

No：  spam (1)
```

- 紧贴索引或者切片的左括号之前。

```
Yes：dct['key'] = lst[index]

No：  dct ['key'] = lst [index]
```

- 为了和另一个赋值语句对齐，在赋值运算符附件加多个空格。

  推荐：

```
x = 1

y = 2

long_variable = 3
```

不推荐：

```
x             = 1
```

```
y               = 2

long_variable = 3
```

Other Recommendations 其他建议

- 避免在尾部添加空格。因为尾部的空格通常都看不见，会产生混乱：比如，一个反斜杠后面跟一个空格的换行符，不算续行标记。有些编辑器不会保留尾空格，并且很多项目（像 CPython）在 pre-commit 的挂钩调用中会过滤掉尾空格。
- 总是在二元运算符两边加一个空格：赋值（=），增量赋值（+=，-=），比较（==,<,>,!=,<>,<=,>=,in,not,in,is,is not），布尔（and, or, not）。
- 如果使用具有不同优先级的运算符，请考虑在具有最低优先级的运算符周围添加空格。有时需要通过自己来判断；但是，不要使用一个以上的空格，并且在二元运算符的两边使用相同数量的空格。

  推荐：

```
i = i + 1

submitted += 1

x = x*2 - 1

hypot2 = x*x + y*y

c = (a+b) * (a-b)
```

不推荐：

```
i=i+1

submitted +=1

x = x * 2 - 1
```

```
hypot2 = x * x + y * y

c = (a + b) * (a - b)
```

- 在制定关键字参数或者默认参数值的时候，不要在=附近加上空格。

  推荐：

```
def complex(real, imag=0.0):

    return magic(r=real, i=imag)
```

不推荐：

```
def complex(real, imag = 0.0):

    return magic(r = real, i = imag)
```

- 功能型注释应该使用冒号的一般性规则，并且在使用->的时候要在两边加空格。

  （参考下面的功能注释得到能够多信息）

  推荐：

```
def munge(input: AnyStr): ...

def munge() -> AnyStr: ...
```

不推荐：

```
def munge(input:AnyStr): ...

def munge()->PosInt: ...
```

- 当给有类型备注的参数赋值的时候，在=两边添加空格（仅针对那种有类型备注和默认值的参数）。

  推荐：

```
def munge(sep: AnyStr = None): ...
```

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

不推荐：

```
def munge(input: AnyStr=None): ...

def munge(input: AnyStr, limit = 1000): ...
```

- 复合语句(同一行中的多个语句)通常是不允许的。

  推荐：

```
if foo == 'blah':

    do_blah_thing()

do_one()

do_two()

do_three()
```

最好别这样：

```
if foo == 'blah': do_blah_thing()

do_one(); do_two(); do_three()
```

- 虽然有时候将小的代码块和 if/for/while 放在同一行没什么问题，多行语句块的情况不要这样用，同样也要避免代码行太长！

  最好别这样：

```
if foo == 'blah': do_blah_thing()

for x in lst: total += x

while t < 10: t = delay()
```

绝对别这样：

```
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)


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

Comments 注释

与代码相矛盾的注释比没有注释还糟，当代码更改时，优先更新对应的注释！

注释应该是完整的句子。如果一个注释是一个短语或句子，它的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！)。

如果注释很短，结尾的句号可以省略。块注释一般由完整句子的一个或多个段落组成，并且每句话结束有个句号。

在句尾结束的时候应该使用两个空格。

当用英文书写时，遵循 Strunk and White （译注：《Strunk and White, The Elements of Style》）的书写风格。

在非英语国家的 Python 程序员，请使用英文写注释，除非你 120%的确信你的代码不会被使用其他语言的人阅读。

Block Comments 块注释

块注释通常适用于跟随它们的某些（或全部）代码，并缩进到与代码相同的级别。块注释的每一行开头使用一个#和一个空格（除非块注释内部缩进文本）。

块注释内部的段落通过只有一个#的空行分隔。

Inline Comments 行内注释

有节制地使用行内注释。

行内注释是与代码语句同行的注释。行内注释和代码至少要有两个空格分隔。注释由#和一个空格开始。

事实上，如果状态明显的话，行内注释是不必要的，反而会分散注意力。比如说下面这样就不需要：

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

- 1

但有时，这样做很有用：

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

- 1

Documentation Strings 文档字符串

编写好的文档说明（也叫“docstrings”）的约定在 PEP 257 中永恒不变。

- 要为所有的公共模块，函数，类以及方法编写文档说明。非公共的方法没有必要，但是应该有一个描述方法具体作用的注释。这个注释应该在 def 那一行之后。
- PEP 257 描述了写出好的文档说明相关的约定。特别需要注意的是，多行文档说明使用的结尾三引号应该自成一行，例如：

```
"""Return a foobang
```

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

"""
```

- 对于单行的文档说明，尾部的三引号应该和文档在同一行。

Naming Conventions 命名规范

Python 库的命名规范很乱，从来没能做到完全一致。但是目前有一些推荐的命名标准。新的模块和包（包括第三方框架）应该用这套标准，但当一个已有库采用了不同的风格，推荐保持内部一致性。

Overriding Principle 最重要的原则

那些暴露给用户的 API 接口的命名，应该遵循反映使用场景而不是实现的原则。

Descriptive: Naming Styles 描述：命名风格

有许多不同的命名风格。这里能够帮助大家识别正在使用什么样的命名风格，而不考虑他们为什么使用。

以下是常见的命名方式：

- b（单个小写字母）
- B（单个大写字母）
- lowercase 小写字母
- lower_case_with_underscores 使用下划线分隔的小写字母
- UPPERCASE 大写字母

- UPPER_CASE_WITH_UNDERSCORES 使用下划线分隔的大写字母
- CapitalizedWords（或者叫 CapWords，或者叫 CamelCase 驼峰命名法 —— 这么命名是因为字母看上去有起伏的外观 [5]）。有时候也被称为 StudlyCaps。

  注意：当在首字母大写的风格中用到缩写时，所有缩写的字母用大写，因此，HTTPServerError 比 HttpServerError 好。
- mixedCase（不同于首字母大写，第一个单词的首字母小写）
- Capitalized_Words_With_Underscores（巨丑无比！）

也有用唯一的短前缀把相关命名组织在一起的方法。这在 Python 中不常用，但还是提一下。比如，os.stat()函数中包含类似以 st_mode，st_size，st_mtime 这种传统命名方式命名的变量。（这么做是为了与 POSIX 系统的调用一致，以帮助程序员熟悉它。）

X11 库的所有公共函数都加了前缀 X。在 Python 里面没必要这么做，因为属性和方法在调用的时候都会用类名做前缀，函数名用模块名做前缀。

另外，下面这种用前缀或结尾下划线的特殊格式是被认可的（通常和一些约定相结合）：

- _single_leading_underscore：（单下划线开头）弱“内部使用”指示器。比如 from M import * 是不会导入以下划线开始的对象的。
- single_trailing_underscore_：（单下划线结尾）这是避免和 Python 内部关键词冲突的一种约定，比如：Tkinter.Toplevel(master, class_=’ClassName’)

- `__double_leading_underscore`：（双下划线开头）当这样命名一个类的属性时，调用它的时候名字会做矫正（在类 FooBar 中，`__boo` 变成了 `_FooBar__boo`；见下文）。
- `__double_leading_and_trailing_underscore__`：（双下划线开头，双下划线结尾）“magic”对象或者存在于用户控制的命名空间内的属性，例如：`__init__`，`__import__`或者`__file__`。除了作为文档之外，永远不要命这样的名。

Prescriptive: Naming Conventions 约定俗成：命名约定

Names to Avoid 应避免的名字

永远不要使用字母‘l’（小写的 L），‘O’（大写的 O），或者‘I’（大写的 I）作为单字符变量名。

在有些字体里，这些字符无法和数字 0 和 1 区分，如果想用‘l’，用‘L’代替。

Package and Module Names 包名和模块名

模块应该用简短全小写的名字，如果为了提升可读性，下划线也是可以用的。Python 包名也应该使用简短全小写的名字，但不建议用下划线。

当使用 C 或者 C++编写了一个依赖于提供高级（更面向对象）接口的 Python 模块的扩展模块，这个 C/C++模块需要一个下划线前缀（例如：_socket）

Class Names 类名

类名一般使用首字母大写的约定。

在接口被文档化并且主要被用于调用的情况下，可以使用函数的命名风格代替。

注意，对于内置的变量命名有一个单独的约定：大部分内置变量是单个单词（或者两个单词连接在一起），首字母大写的命名法只用于异常名或者内部的常量。

Exception Names 异常名

因为异常一般都是类，所有类的命名方法在这里也适用。然而，你需要在异常名后面加上“Error”后缀（如果异常确实是一个错误）。

Global Variable Names 全局变量名

（我们希望这一类变量只在模块内部使用。）约定和函数命名规则一样。

通过 from M import * 导入的模块应该使用 **all** 机制去防止内部的接口对外暴露，或者使用在全局变量前加下划线的方式（表明这些全局变量是模块内非公有）。

Function Names 函数名

函数名应该小写，如果想提高可读性可以用下划线分隔。

大小写混合仅在为了兼容原来主要以大小写混合风格的情况下使用（比如threading.py），保持向后兼容性。

Function and method arguments 函数和方法参数

始终要将 self 作为实例方法的的第一个参数。

始终要将 cls 作为类静态方法的第一个参数。

如果函数的参数名和已有的关键词冲突，在最后加单一下划线比缩写或随意拼写更好。因此 class_ 比 clss 更好。（也许最好用同义词来避免这种冲突）

Method Names and Instance Variables 方法名和实例变量

遵循这样的函数命名规则：使用下划线分隔小写单词以提高可读性。

在非共有方法和实例变量前使用单下划线。

通过双下划线前缀触发 Python 的命名转换规则来避免和子类的命名冲突。

Python 通过类名对这些命名进行转换：如果类 Foo 有一个叫 `__a` 的成员变量， 它无法通过 `Foo.__a` 访问。（执着的用户可以通过 `Foo._Foo__a` 访问。）一般来说，前缀双下划线用来避免类中的属性命名与子类冲突的情况。

注意：关于`__names` 的用法存在争论（见下文）。

Constants 常量

常量通常定义在模块级，通过下划线分隔的全大写字母命名。例如：

MAX_OVERFLOW 和 TOTAL。

Designing for inheritance 继承的设计

始终要考虑到一个类的方法和实例变量（统称：属性）应该是共有还是非共有。如果存在疑问，那就选非共有；因为将一个非共有变量转为共有比反过来更容易。

公共属性是那些与类无关的客户使用的属性，并承诺避免向后不兼容的更改。非共有属性是那些不打算让第三方使用的属性；你不需要承诺非共有属性不会被修改或被删除。

我们不使用“私有（private）”这个说法，是因为在 Python 中目前还没有真正的私有属性（为了避免大量不必要的常规工作）。

另一种属性作为子类 API 的一部分（在其他语言中通常被称为“protected”）。有些类是专为继承设计的，用来扩展或者修改类的一部分行为。当设计这样的类时，要谨慎决定哪些属性时公开的，哪些是作为子类的 API，哪些只能在基类中使用。

贯彻这样的思想，一下是一些让代码 Pythonic 的准则：

- 公共属性不应该有前缀下划线。

- 如果公共属性名和关键字冲突，在属性名之后增加一个下划线。这比缩写和随意拼写好很多。（然而，尽管有这样的规则，在作为参数或者变量时，‘cls’是表示‘类’最好的选择，特别是作为类方法的第一个参数。）

  注意 1：参考之前的类方法参数命名建议

- 对于单一的共有属性数据，最好直接对外暴露它的变量名，而不是通过负责的 存取器（accessor）/突变（mutator） 方法。请记住，如果你发现一个简单的属性需要成长为一个功能行为，那么 Python 为这种将来会出现的扩展提供了一个简单的途径。在这种情况下，使用属性去隐藏属性数据访问背后的逻辑。

  注意 1：属性只在 new-style 类中起作用。

  注意 2：尽管功能方法对于类似缓存的负面影响比较小，但还是要尽量避免。

  注意 3：属性标记会让调用者认为开销（相当的）小，避免用属性做开销大的计算。

- 如果你的类打算用来继承的话，并且这个类里有不希望子类使用的属性，就要考虑使用双下划线前缀并且没有后缀下划线的命名方式。这会调用 Python 的命名转换算法，将类的名字加入到属性名里。这样做可以帮助避免在子类中不小心包含了相同的属性名而产生的冲突。

  注意 1：只有类名才会整合进属性名，如果子类的属性名和类名和父类都相同，那么你还是会有命名冲突的问题。

  注意 2：命名转换会在某些场景使用起来不太方便，例如调试，`__getattr__()`。然而命名转换的算法有很好的文档说明并且很好操作。

  注意 3：不是所有人都喜欢命名转换。尽量避免意外的名字冲突和潜在的高级调用。

Public and internal interfaces 公共和内部的接口

任何向后兼容保证只适用于公共接口，因此，用户清晰地区分公共接口和内部接口非常重要。

文档化的接口被认为是公开的，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该是内部的。

为了更好地支持内省（introspection），模块应该使用`__all__`属性显式地在它们的公共 API 中声明名称。将`__all__`设置为空列表表示模块没有公共 API。

即使通过`__all__`设置过，内部接口（包，模块，类，方法，属性或其他名字）依然需要单个下划线前缀。

如果一个命名空间（包，模块，类）被认为是内部的，那么包含它的接口也应该被认为是内部的。

导入的名称应该始终被视作是一个实现的细节。其他模块必须不能间接访问这样的名称，除非它是包含它的模块中有明确的文档说明的 API，例如 os.path 或者是一个包里从子模块公开函数接口的 `__init__` 模块。

Programming Recommendations 编程建议

- 代码应该用不损害其他 Python 实现的方式去编写（PyPy，Jython，IronPython，Cython，Psyco 等）。

  比如，不要依赖于在 CPython 中高效的内置字符连接语句 a += b 或者 a = a + b。这种优化甚至在 CPython 中都是脆弱的（它只适用于某些类型）并且没有

出现在不使用引用计数的实现中。在性能要求比较高的库中，可以种 " .join() 代替。这可以确保字符关联在不同的实现中都可以以线性时间发生。

- 和像 None 这样的单例对象进行比较的时候应该始终用 is 或者 is not，永远不要用等号运算符。

  另外，如果你在写 if x 的时候，请注意你是否表达的意思是 if x is not None。举个例子，当测试一个默认值为 None 的变量或者参数是否被设置为其他值的时候。这个其他值应该是在上下文中能成为 bool 类型 false 的值。

- 使用 is not 运算符，而不是 not ... is 。虽然这两种表达式在功能上完全相同，但前者更易于阅读，所以优先考虑。

  推荐：

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

不推荐：

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

- 当使用富比较（rich comparisons，一种复杂的对象间比较的新机制，允许返回值不为-1,0,1）实现排序操作的时候，最好实现全部的六个操作符（`__eq__`, `__ne__`, `__lt__`, `__gt__`, `__ge__`）而不是依靠其他的代码去实现特定的比较。

  为了最大程度减少这一过程的开销， functools.total_ordering() 修饰符提供了用于生成缺少的比较方法的工具。

  PEP 207 指出 Python 实现了反射机制。因此，解析器会将 y > x 转变为 x < y，将 y >= x 转变为 x <= y，也会转换 x == y 和 x != y 的参数。sort() 和

min()方法确保使用<操作符，max()使用>操作符。然而，最好还是实现全部六个操作符，以免在其他地方出现冲突。

- 始终使用 def 表达式，而不是通过赋值语句将 lambda 表达式绑定到一个变量上。

  推荐：

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

不推荐：

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

第一个形式意味着生成的函数对象的名称是“f”而不是泛型“< lambda >”。这在回溯和字符串显示的时候更有用。赋值语句的使用消除了 lambda 表达式优于显式 def 表达式的唯一优势（即 lambda 表达式可以内嵌到更大的表达式中）。

- 从 Exception 继承异常，而不是 BaseException。直接继承 BaseException 的异常适用于几乎不用来捕捉的异常。

  设计异常的等级，要基于扑捉异常代码的需要，而不是异常抛出的位置。以编程的方式去回答“出了什么问题？”，而不是只是确认“出现了问题”（内置异常结构的例子参考 PEP 3151 ）

  类的命名规范适用于这里，但是你需要添加一个“Error”的后缀到你的异常类，如果异常是一个 Error 的话。非本地流控制或者其他形式的信号的非错误异常不需要特殊的后缀。
- 适当地使用异常链接。在 Python 3 里，为了不丢失原始的根源，可以显式指定“raise X from Y”作为替代。

当故意替换一个内部异常时（Python 2 使用“raise X”， Python 3.3 之后 使用 “raise X from None”），确保相关的细节转移到新的异常中（比如把 AttributeError 转为 KeyError 的时候保留属性名，或者将原始异常信息的文本内容内嵌到新的异常中）。

- 在 Python 2 中抛出异常时，使用 rasie ValueError（‘message’）而不是用老的形式 raise ValueError, ‘message’。

  第二种形式在 Python3 的语法中不合法

  使用小括号，意味着当异常里的参数非常长，或者包含字符串格式化的时候，不需要使用换行符。

- 当捕获到异常时，如果可以的话写上具体的异常名，而不是只用一个 except: 块。

  比如说：

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

如果只有一个 except: 块将会捕获到 SystemExit 和 KeyboardInterrupt 异常，这样会很难通过 Control-C 中断程序，而且会掩盖掉其他问题。如果你想捕获所有指示程序出错的异常，使用 except Exception: （只有 except 等价于 except BaseException:）。

两种情况不应该只使用‘excpet’块：

1. 如果异常处理的代码会打印或者记录 log；至少让用户知道发生了一个错误。

2. 如果代码需要做清理工作，使用 raise..try...finally 能很好处理这种情况并且能让异常继续上浮。

    1. 当给捕捉的异常绑定一个名字时，推荐使用在 Python 2.6 中加入的显式命名绑定语法：

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

为了避免和原来基于逗号分隔的语法出现歧义，Python3 只支持这一种语法。

- 当捕捉操作系统的错误时，推荐使用 Python 3.3 中 errno 内定数值指定的异常等级。
- 另外，对于所有的 try/except 语句块，在 try 语句中只填充必要的代码，这样能避免掩盖掉 bug。

  推荐：

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

不推荐：

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

- 当代码片段局部使用了某个资源的时候，使用 with 表达式来确保这个资源使用完后被清理干净。用 try/finally 也可以。
- 无论何时获取和释放资源，都应该通过单独的函数或方法调用上下文管理器。举个例子：

  推荐：

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

不推荐：

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

第二个例子没有提供任何信息去指明`__enter__`和`__exit__`方法在事务之后做出了关闭连接之外的其他事情。这种情况下，明确指明非常重要。

- 返回的语句保持一致。函数中的返回语句都应该返回一个表达式，或者都不返回。如果一个返回语句需要返回一个表达式，那么在没有值可以返回的情况下，需要用 return None 显式指明，并且在函数的最后显式指定一条返回语句（如果能跑到那的话）。

  推荐：

```
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：

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

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

- 使用字符串方法代替字符串模块。

  字符串方法总是更快，并且和 unicode 字符串分享相同的 API。如果需要兼容 Python2.0 之前的版本可以不用考虑这个规则。

- 使用 ” .startswith() 和 ” .endswith() 代替通过字符串切割的方法去检查前缀和后缀。

  startswith()和 endswith()更干净，出错几率更小。比如：

```
推荐: if foo.startswith('bar'):

糟糕: if foo[:3] == 'bar':
```

- 对象类型的比较应该用 isinstance()而不是直接比较 type。

```
正确: if isinstance(obj, int):


糟糕: if type(obj) is type(1):
```

当检查一个对象是否为 string 类型时，记住，它也有可能是 unicode string！在 Python2 中，str 和 unicode 都有相同的基类：basestring，所以你可以这样：

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

注意，在 Python3 中，unicode 和 basestring 都不存在了（只有 str）并且 bytes 类型的对象不再是 string 类型的一种（它是整数序列）

- 对于序列来说（strings，lists，tuples），可以使用空序列为 false 的情况。

```
正确: if not seq:
      if seq:


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

- 书写字符串时不要依赖单词结尾的空格，这样的空格在视觉上难以区分，有些编辑器会自动去掉他们（比如 reindent.py （译注：re indent 重新缩进））

- 不要用 == 去和 True 或者 False 比较：

```
正确: if greeting:

糟糕: if greeting == True:

更糟: if greeting is True:
```

Function Annotations 功能注释

随着 PEP 484 的引入，功能型注释的风格规范有些变化。

- 为了向前兼容，在 Python3 代码中的功能注释应该使用 PEP 484 的语法规则。（在前面的章节中对注释有格式化的建议。）
- 不再鼓励使用之前在 PEP 中推荐的实验性样式。
- 然而，在 stdlib 库之外，在 PEP 484 中的实验性规则是被鼓励的。比如用 PEP 484 的样式标记大型的第三方库或者应用程序，回顾添加这些注释是否简单，并观察是否增加了代码的可读性。
- Python 的标准库代码应该保守使用这种注释，但新的代码或者大型的重构可以使用这种注释。
- 如果代码希望对功能注释有不同的用途，建议在文件的顶部增加一个这种形式的注释：

```
# type: ignore
```

这会告诉检查器忽略所有的注释。（在 PEP 484 中可以找到从类型检查器禁用投诉的更细粒度的方法。）

- 像 linters 一样，类型检测器是可选的可独立的工具。默认情况下，Python 解释器不应该因为类型检查而发出任何消息，也不应该基于注释改变它们的行为。

- 不想使用类型检测的用户可以忽略他们。然而，第三方库的用户可能希望在这些库上运行类型检测。为此， PEP 484 建议使用存根文件类型：.pyi 文件，这种文件类型相比于.py 文件会被类型检测器读取。存根文件可以和库一起，或者通过 typeshed repo[6] 独立发布（通过库作者的许可）
- 对于需要向后兼容的代码，可以以注释的形式添加功能型注释。参见 PEP 484 的相关部分 [7]。

参考

---

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. 挂行缩进是一种类型设置样式，其中除第一行之外，段落中的所有行都缩进。在 Python 中，这个术语是用来描述一种风格：在被括号括起来的语句中，左括号是这一行最后一个非空格字符，随后括号内的内容每一行进行缩进，直到遇到右括号。
4. Donald Knuth’ s The TeXBook, pages 195 and 196
5. http://www.wikipedia.com/wiki/CamelCase
6. Typeshed repo https://github.com/python/typeshed
7. 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