PEP 8 -- Python 代码风格指南

1、引言

本文档给出了 Python 编码规约，主要 Python 发行版中的标准库即遵守该规约。对于 C 代码风格的 Python 程序，请参阅配套的 C 代码风格指南。

本文档和 PEP 257（文档字符串规约）是根据 Guido 的 Python 风格指南一文改编的，其中还增加了 Barry 提出的一些风格指南。

风格指南并非一成不变，它本身也在不断发展，过去的惯例会因语言本身的变化而过时。

许多项目有自己的编码风格，在发生冲突时，这类项目的风格指南应优先考虑。

2、固执己见必成心灵之魔

Guido 认为：代码被阅读之频繁远甚于其被编写。所以，这里提供的准则旨在提高代码的可读性并使其在各种 Python 代码中保持一致。如 PEP 20 所述，“可读性至关重要”。

风格指南是有关一致性的重要规约。书写与此风格指南一致的代码很重要，书写与项目风格一致的代码更加重要，在一个模块或函数内, 书写风格一致的代码超级重要。

所以，风格指南并不适用于所有情况，要知道什么时候不一致，这需要你运用自己的智慧进行判断。多看一些例子，选择你认为最好的那个，必要的时候要懂得向别人请教。

尤其注意：**不要为了满足风格指南而破坏与过去代码风格的兼容性！**

> *以下情况可以不必考虑风格指南：*
>
> 1. *即使对于习惯此风格代码的人，应用此风格指南也会使代码的可读性降低。*
> 2. *为了与上下文代码风格一致(可能由于历史原因，上下文的风格也违背了本指南)，当然这也是一个规范代码风格的机会(通过改变上下文的风格)。*
> 3. *有些代码早于此风格指南出现并且没有其他理由要修改这些代码。*
> 4. *代码需要与旧版本的 Python 代码兼容，但旧版本不支持此风格指南。*

3、代码布局

(1)缩进

每级缩进对应四个空格。

()、[]或者{}可以隐式的换行，三种括号所包裹的代码要么垂直对齐，要么悬挂缩进。当使用悬挂缩进时，应该注意：参数不能放在首行，续行要再缩进一级以便和后边的代码区别开。

> *正确写法：*
> *#与左括号对齐（垂直对齐写法）*

```
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

> *#悬挂缩进*

```
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

> *#如果后边有代码行，悬挂缩进增加一级*

```
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

> *错误写法：*
> *#参数放在第一行而没有垂直对齐*

```
foo = long_function_name(var_one, var_two,
    var_three, var_four)
```

> *#后边有代码行时没有增加缩进*

```
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

> *对于续行来说，4个空格的缩进规则可以不必遵守*
> *#可以不采用是4个空格的缩进方法*

```
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

当 if 语句的条件部分很长以至于需要将其写成多行时，需要注意，if 和单个空格以及左括号正好是 4 个空格缩进，这可能与嵌套在if语句中的代码块产生冲突。本文档不提供确切的方法来解决条件行和 if 语句的嵌套代码块的冲突。以下是一些可行的处理方法，但不必局限于此：

> *#无额外的缩进*

```
if (this_is_one_thing and
    that_is_another_thing):
    do_something()
```

> *#通过注释进行区分*

```
if (this_is_one_thing and
    that_is_another_thing):
```

```
    # Since both conditions are true, we can frobnicate.
    do_something()
```

#对条件行的续行增加缩进

```
if (this_is_one_thing
        and that_is_another_thing):
    do_something()
```

(另请参阅下面的关于在二元运算符之前还是之后换行的讨论。)

多行结构的右括号（包括圆括号，中括号和花括号）可以置于最后一行代码的第一个非空白字符下面，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，也可以放在多行结构的下一行的第一个字符位置，如下所示：

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

（2）“Tab”还是空格?

推荐使用空格控制缩进。

tab 只用于与之前的代码保持一致的情况。

Python3 不允许混合使用 tab 和空格控制缩进。

Python2 中混合使用 tab 和空格缩进的代码应改成只使用空格缩进。

当使用 -t 选项调用 Python2 命令行解释器时，会给出混合使用 tab 和空格的警告。使用 -tt 选项时，这些警告会变为错误。我们推荐使用这些选项。

（3）单行最大长度

每行最多 79 个字符。

对于具有较少结构限制的长文本块（如文档字符串和注释）每行最多 72 个字符。

限制编辑器的窗口宽度可以在屏幕上并排打开多个文件，同时也能更好地使用代码审阅工具（在相邻两列中显示两个版本的代码）。

大多数工具默认的换行方式破坏了代码可见的结构，让代码变得难于理解。在窗口大小为 80 的编辑器中，即使换行的代码会在最后一列被打上标记，为了避免自动换行也需要限制每行字符长度。一些基于 Web 的工具可能根本不会自动换行。

有些团队会喜欢很长的代码行。如果代码只由或主要由他们维护并且也在内部达成一致的话，将代码行的长度增加到 80 到 100 个字符（实际上最大行长是 99 个字符）都没有什么问题。当然，注释和文档字符串仍然维持在72个字符以内。

Python 标准库比较保守，选择将代码行最大长度限制为 79 个字符，注释或者文档字符串限制为 72 个字符。

Python 中首选的换行方式是使用括号括起来隐式地换行。使用括号可以将长的代码行分成多个较短行。这种写法要优先于使用 \ 换行。

在某些情况下，也可以使用 \ 换行，比如，当 with 语句很长，隐式换行又不方便，就可以使用 \ 。

```
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())
```

(参阅前面有关多行 if 语句的讨论，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。

（4）在二元运算符之前还是之后换行?

长期以来，一直推荐的风格是在二元运算符之后换行，但是这会影响代码可读性，一是会使运算符分散在屏幕的不同列上，二是会使每个运算符留在前一行，并远离操作数分离。必须人为地判断应该加上或者减去哪些东西，这就增加了人眼的负担。

> *#错误写法:*
> *运算符远离操作数*

```
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

为了解决这一可读性问题，数学家和出版商遵循了相反的规约。Donald Knuth 在他的《Computers and Typesetting》丛书中解释了这一规约。“虽然段落中的公式总是在二元运算符后换行，但显示公式时总是在二元运算符之前换行。”

遵循数学上的传统可以写出可读性更好的代码。

> *#正确写法:*
> *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，只要前后保持一致，在二元操作符之前或之后换行都可以。对于新写的代码，建议使用 Knuth 推荐的风格。

（5）空行

顶级函数和类的定义前后空两行。

类内部的方法定义前后空一行。

可以使用额外的空行（尽量少）来分隔相关函数组。一系列相关的仅占一行的函数之间的空白行可以省略（比如一系列虚函数）。

在函数内部可以使用空行（尽量少）来分割逻辑上的代码块。

Python 将 Ctrl + L 换页符作为空格，但许多工具将它当作分页符，所以你可以用这种方法把文件中的相关代码放在一页。注意：一些编辑器和基于 Web 的代码阅读器可能不会把 Ctrl + L 当作换页符处理，并且在该位置会显示其他的字符。

（6）源文件编码

当前核心的 Python 发行版本一直使用 utf-8 编码，在 Python2 中则使用 ASCII 编码。

Python2 使用 ASCII ， Python3 使用 utf-8 编码，这在文件中不需要进行编码声明。

在标准库中，不使用非默认编码，除非出于测试目的。当注释或文档字符串中使用的作者名包含非 ASCII 字符，也可以使用非默认编码。其他情况下，若要在字符串中包含非 ASCII 数据，建议使用转义字符 `\x` `\u` `\U` `\N` 。

对 Python3 及更高版本的标准库有以下规定（详见 PEP 3131）： Python 标准库中的所有标识符均为 ASCII 编码并且尽可能使用英文单词。某些情况也可以使用非英文的缩写和术语。另外，字符串字面量和注释也必须使用 ASCII 字符。以下情况例外：

- 测试非 ASCII 字符的特性
- 作者姓名。作者姓名如果不在拉丁字母表中（latin-1, ISO/IEC 8859-1 字符集），必须给出基于此字母表的音译名。

我们鼓励全球受众的开源项目采用类似规约。

（7）导入

- 导入语句应该分开写，而不是都放到一行，例如：

  > *正确写法：*

  ```
  import os
  import sys
  ```

  > *错误写法：*

  ```
  import sys, os
  ```

这种写法也可以：

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

- 导入语句一般放在文件开头，在模块注释和文档字符串之后，模块全局变量和常量之前。

  > *注意分组导入，并遵循一下顺序：*
  >
  > > *1. 标准库导入*
  > > *2. 相关的第三方包导入*
  > > *3. 本地应用或库的特定导入*
  >
  > *不同的组应使用空行分隔*

- 推荐绝对路径导入方式，通常这样更有可读性，在系统配置错误时的情况下（比如包里的文件路径有以 sys.path 结尾的。），也会表现得更好（至少会给出更好的错误提示信息）：

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

  然而，清晰的相对路径导入方式也可以接受，特别是当包的结构比较复杂，使用绝对路径导入方式没有必要，反而会使代码显得冗杂。

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

  标准库的代码没有复杂的包结构，并且一直使用绝对路径的导入方式。

  不能使用隐式的相对路径导入，这在 Python3 中已经被移除。

- 当从包含类的模块中导入类时，可以使用以下写法：

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

  如果这种写法导致和本地命名空间冲突，则改为：

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

  然后使用 `myclass.MyClass` 或者 `foo.bar.yourclass.YourClass`。

- 避免使用通配符的方式导入（ `from <module> import *` ），因为这会使当前命名空间中的名称含义不清晰，给读者和许多自动化工具造成困扰。有一种情况正当使用通配符导入的情形，就是将一个内部接口重新发布作为公共 API （比如，使用可选的加速模块中的定义覆盖纯 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名称时，指南后面关于公共和内部接口的部分仍然适用。

（8）模块级双下划线名称

模块级的双下划线名称（即在名称的开始和结尾处都有两条下划线）如__all__，**author__，__version** 等，应该写在文档字符串之后，除 `from __future__` 外其他导入语句之前。Python 要求在模块中 `future-imports` 语句要放在除文档字符串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
```

4、字符串引用

在 Python 中，单引号字符串和双引号字符串是一样的。本 PEP 文档在这一点上不做要求，选用一种并坚持下去就好。当字符串本身包含单引号或者双引号时，那就选用未包含的引号来表示字符串，这样可以避免使用 `\`，代码也更易读。

如果使用三引号形式的字符串，则用双引号 `"` 组成三引号 `"""`，这样可以和 PEP 257 中的文档字符串规约保持一致。

5、表达式和语句中的空格

（1）一些痛点

以下情形中要避免使用过多空格：

- 括号内部紧挨括号的地方，包括圆括号、方括号和花括号。

  > *正确写法：*

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

  > *错误写法：*

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

- 逗号后面跟着右括号。

  > *正确写法：*

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

> *错误写法：*

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

- 逗号，分号，和冒号之前。

  > *正确写法：*

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

  > *错误写法：*

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

- 然而，在切片操作中，冒号作为二元运算符，两边应该具有相同数目的空格（可将其视作最低优先级的运算符）。在扩展切片操作中，两个冒号左右两边的空格数都应该相等。当然，如果切片操作省略了参数，那么空白也应该省略。

  > *正确写法：*

```
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]
```

  > *错误写法：*

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

- 函数调用时，包裹参数列表的左括号之前：

  > *正确写法：*

```
spam(1)
```

  > *错误写法：*

```
spam (1)
```

- 在索引或者切片操作紧挨着的左括号前边：

> *正确写法：*

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

> *错误写法：*

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

- 为了对齐，在赋值号或者其他运算符前后加多个空格。

> *正确写法：*

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

> *错误写法：*

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

(2) 其他建议

- 避免任何行末空格，因为通常不可见，它们容易让人困惑。例如，`\` 后跟一个空格和一个新行，`\` 将不会被当作续行符。有些编辑器可以自动去除行末空格。对于这种情况，许多项目（如CPython本身）会用钩子做提交前检查。
- 这些二元运算符两边通常保留一个空格：赋值运算符（=），增强的复制运算符（+=，-=等）关系运算符（==，<，>，!=，<=，>=，in，not in，is，is not），布尔运算符（and，or，not）。
- 如果混用了不同优先级的运算符，可以考虑在低优先级运算符两边增加空格。但不要超过一个，并且要保持二元运算符两边的空格数量相同。

> *正确写法：*

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

> *错误写法：*

```
i=i+1
submitted +=1
x = x * 2 - 1
```

```
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

- 当表示关键字参数或者默认参数时，等号两边不加空格。

  正确写法：

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

  错误写法：

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

- 函数注释中的冒号遵循一般的加空格的规则，如果有箭头，要在其两边加空格。（更多信息请见下面的函数注释。）

  正确写法：

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

  错误写法：

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

- 当组合使用参数注释和参数默认值时，赋值号两边要加空格（但仅对既有空格又有默认值的参数成立）。

  正确写法：

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

  错误写法：

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

- 通常不鼓励使用复合语句（即将多条语句写在一行）。

  正确写法：

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

```
do_two()
do_three()
```

> *最好不要这样写：*

```

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

- 然而，小型的 `if/for/whiel` 语句放在一行是可以的。但是有多条分句时不要这样做。也要避免无谓的换行！

> *最好不要这样写：*

```

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

> *绝对不要这样写：*

```

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()
```

6、何时在末尾加逗号

末尾的逗号通常是可选的。但是，在定义单元素元组时是必须的（而且在 Python2 中，逗号对 print 语句有特殊语义）。清楚起见，建议使用括号括起来（在技术上是冗余的）。

> *正确写法：*

```

FILES = ('setup.cfg',)
```

> *也可以，但是难于理解：*

```

FILES = 'setup.cfg',
```

当使用版本控制系统时，在将来有可能扩展的值和参数列表或者导入条目的末尾添加冗余的逗号是有好处的。书写模式：每行只写一个值并且加上逗号，在最后的新行写上右括号。但是，把逗号和右括号写在同一行毫无意义（除了上面提到的单元素元组）。

> *正确写法：*

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

> *错误写法：*

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

7、注释

和代码相矛盾的注释还不如没有注释。当代码更新时，要优先更改注释，使其保持最新状态。

注释应该是完整的句子。第一个单词首字母要大写，除非是一个小写字母开头的标识符（永远不要修改标识符的大小写！）。

块注释通常由完整句子构成的一个或多个段落组成，每个句子都以句号结束。

在多语句注释中，除了最后一条句子，应当在句尾的句号后面加两个空格。

如果使用英文写作，参考 Strunk 和 White 的著作。

来自非英语国家的 Python 程序员们：请书写英文注释，除非你 120% 确定你所写的代码永远不会被不懂你所用语言的人读到。

（1）块注释

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。块注释以一个 # 和一个空格打头（除非注释内的文本也有缩进）。

块注释中的段落用以单个 # 字符开头的空行分隔。

（2）行内注释

少用行内注释。

行内注释和代码语句写在同一行，至少以两个空格分隔。并且也以一个 # 和一个空格打头。

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。不要有以下写法：

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

但有时却是必要的：

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

(3)文档字符串

要写出好的文档字符串（又名"docstrings"），请参阅 PEP 257 。

- 应该为所有的公共模块、函数、类和方法编写文档字符串。文档字符串对于非公共方法不是必须的，但是应该留有注释以说明此方法的用途，此注释要放在 `def` 语句的下一行。
- PEP 257 有好文档字符串规约。尤其注意，多行文档字符串结尾处的 `"""` 要单独占一行，例如：

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

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

- 对于单行文档字符串，可以把末尾的 `"""` 放在同一行。

8、命名规约

Python 库的命名规约有些混乱，所以我们无法就此保持完全一致。但我们当前还是有一些值得推荐的命名规约。书写新的模块和包（包括第三方框架）时，应当遵循这些标准。但是如果现有的库遵循了不同的代码风格，那么应该保持内部代码的一致性。

（1）首要原则

对用户可见的公共部分 API ，其名称应该反应出其用途而不是实现。

（2）描述：命名风格

不同的命名风格有很多，最好能从应用他们的代码而识别出对应命名风格。

注意区别以下命名风格：

- b（单个小写字符）
- B（单个大写字符）
- lowercase（小写单词）
- lower_case_with_underscores（下划线连接的小写单词）
- UPPERCASE（大写单词）
- UPPER_CASE_WITH_UNDERSCORES（下划线连接的大写单词）
- CapitalizedWords（也叫做CapWords或CamelCase – 因为单词首字母大写看起来像驼峰而得名）有时也被称为 StudlyCaps 。

  注意：当使用首字母缩略词时，将缩略词的所有字母大写。因此HTTPServerError的写法比HttpServerError更好。
- mixedCase（和CapitalizedWords的不同之处是最开始的字母要小写！）
- Capitalized_Words_With_Underscores（真丑！）

还有的命名风格用简短的唯一前缀将相关的名称写在一组。这在 Python 中不常用，但完整起见，这里点出来。例如， `os.stat()` 函数会返回一个元组，其中包含 `st_mode` ， `st_size` ， `st_mtime` 等名称。这样做是为了强调和 POSIX 系统调用结构之间的关系，让程序员更熟悉。

X11 库中的公共的函数名都以 X 开头。在 Python 中，一般认为这种风格没什么必要，因为属性和方法名都以对象名为前缀，函数名以模块名为前缀。

另外，以下用下划线开始或结尾的特殊形式也是被认可的（与其他规约结合使用）：

- _single_leading_underscore：内部使用的弱标识。例如，`from M import *` 语句并不会导入以下划线开头的对象。
- single_trailing_underscore_：以单下划线结尾避免和 Python 关键字冲突。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时触发命名修饰（在 FooBar 类内部，__boo 命名会被修饰成_Foo__boo；见以下）。
- __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中。例如：`__init__`，`__import__` 或者 `__file__`。除了按文档描述来使用这些命名，不要私自发明使用。

（3）规范：命名规约

避免的命名

不要使用小写 `l`（el），大写 `O`（oh）或者大写 `I`（eye）作为单字符变量名。某些字体中，这些字符和数字0、1无法区分。如果想用 `l`，可以用 `L` 代替。

ASCII 兼容性

如 PEP 3131 policy 部分所述，标准库中的标识符必须与 ASCII 兼容。

包和模块命名

模块名要简短并且全部小写。如果能提高可读性，也可以在模块名中加下划线。Python 包名称也要简短和小写，但不鼓励使用下划线。当使用 C 或 C++ 写的扩展模块附带 Python 模块，并且该模块提供更高级的接口（比如，更具面向对象特性）时，则 C/C++ 模块名以下划线开头（例如，_sociket）。

类命名

类命名应使用驼峰命名法。

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规约。注意，内建名称有独立的命名规约：大部分内建名称是一个单词（或者组合使用的两个单词），驼峰命名法只适用于异常名和内建常量。

类型变量命名

类型变量名在 PEP 484 中引入，一般使用驼峰命名法，且要尽量简短：`T` `AnyStr` `Num`。对应用到协变和逆变行为的相应变量名，建议添加后缀 `_co` 或 `_contra`。例如：

```
from typing import TypeVar
```

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

异常命名

因为异常也是类，所以类命名规约也使用于异常。但是，如果异常实际上是抛出错误时，那么异常名后应该加上"Error"。

全局变量命名

（我们希望这些变量只能用于一个模块中。）这些规约和函数命名规约大致相同。

对于使用方式设计成 `from M import *` 的模块，应该使用__all__机制来避免导入全局变量。或者采用在全局变量前加下划线的旧规约，来说明这些变量是模块级非公共的。

函数和变量名

函数名应该小写，必要时可使用下划线分隔单词来提高可读性。

变量名和函数名遵循相同的规约。

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

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` 。

当函数的参数名称与保留关键字冲突时，相比使用缩写或拼写简化，使用以下划线结尾的名称更好。所以 `class_` 比 `clss` 更好。（或许更好的方法是通过使用同义词来避免这种冲突。）

方法命名和实例变量

使用函数命名规约：单词小写，必要时以下划线分隔。

非公共方法和实例变量以下划线打头。

为避免和子类产生命名冲突，使用双下划线打头的方式来调用 Python 的命名修饰机制。

Python 将这种名称和类名混合到一起使用：如果类 `Foo` 有一个属性叫 `__a` ，则不能使用 `Foo.__a` 的方式访问。（如果用户坚持想访问，可以使用 `Foo._Foo__a` 。）一般来说，使用双下划线打头仅仅是为了避免与可继承类的属性发生命名冲突。

注意：关于双下划线开头的命名方式还有争议。

常量

常量通常是在模块级别定义的，并且全部采用大写字母，单词之间以下划线分隔。例如： `TOTAL` `MAX_OVERFLOW` 。

继承的设计

永远记得区别类方法和实例变量（属性）应该是公开的还是非公开的。如果有所疑虑，就选择非公开。因为，将非公开属性变成公开属性和容易，反之很难。

类中的公开属性是留给与此类无关的客户使用的，并且保证向后的兼容性。非公开属性是那些不打算让第三方使用的部分，这类属性可能被更改甚至移除。

这里，我们不使用"私有的"这一术语，因为在 Python 中没有真正的私有属性（避免了大量不必要的工作）。

另一类属性是子类 API （在其他语言中经常被称为"受保护的"）的一部分。有些类是为继承而设计的，要么会扩展要么会修改类的行为。当设计这样的类时，注意，一定要明确的说明哪些属性是公开的，哪些是子类 API，哪些是真正只被基类调用的。

考虑到这些，得出以下 Python 风格指南：

- 公开属性不应该以下划线打头。
- 如果你的公开属性名与保留关键字冲突，那就在属性名后增加一条下划线。这比采用缩略词或简写方式更好。（然而，虽说有了这条规则，对于任何类变量或参数，尤其是类方法的第一个参数，`cls` 都是更好的选择，因为这是通识。）

  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，最好只公开属性名，不要有访问器和修改器。请记住，如果您需要对简单的数据属性增加功能行为，Python 为功能增强提供了一条简单的途径。这种情况下使用 `properties` 注解把功能实现隐藏在简单数据属性访问语法之后。

  注意 1：`properties` 注解仅对新式类起作用。

  注意 2：尽量消除功能行为的副作用，尽管缓存之类的副作用没有什么坏处。

  注意 3：避免对计算量大的操作使用 `properties` 注解，属性注解会让调用者认为开销相对较低。
- 如果你的类会被子类继承，但有些属性你又不想让子类使用，可以考虑用双下划线打头结尾没有下划线的方式命名。这样会调用 Python 的命名修饰算法，将类名修饰添加到属性名中。这样可以避免属性命名冲突，因为子类可能在不经意间设置同名的属性。

  注意 1：命名修饰仅仅是简单的用类名修饰了属性名，如果你在子类中同时使用类名+属性名，也会遇到命名冲突问题。

  注意 2：命名修饰可以有特定的用途，比如调试和 `__getattr__()` ，只是不够方便。然而，命名修饰算法已经有很好的文档化了，手动执行也很容易。

  注意 3：不是所有人都喜欢命名修饰，要尽力在避免命名冲突和方便调用之间寻求一种平衡。

公开的和内部的接口

任何向后兼容性保证仅适用于公开接口。因此，用户必须能够明确区分公开和内部接口。一般认为文档化的接口是公开的，除非文档明确说明他们是临时的或者是内部的接口，那就不保证向后兼容性。所有的非文档化接口应当被认为是内部的。

为了更好的审视公开接口和内部接口，模块应该在 `__all__` 属性中明确声明哪些是公开的 API。将 `__all__` 设置为空列表以表明模块中没有公开 API。

即使正确设置了 `__all__` 属性，内部接口（包，模块，类，函数，属性或者其他名称）也应该使用一条下划线打头。

在任何内部的命名空间（包、模块或类）中的接口也被认为是内部的。

应始终将导入的名称视为底层实现。其他模块不能依赖对这些导入名称的间接访问，除非在包含的模块中已经被明确地文档化了，比如 `os.path` 或者包的 `__init__` 模块，他们使用子模块中的公开功能。

9、编程建议

- 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 CPython 对字符串拼接的高效实现（即 `a += b` 或者 `a = a + b` 这样的语句）。即使在 CPython 中这种优化方式也很脆弱，它仅对某些类型起作用，甚至在不使用引用计数方法的实现中都不存在。库中性能敏感的部分应该用 `''.join()` 替代。这可以保证，在各种不同的实现中，拼接操作都是线性时间的。

- 与 None 这样的单例做比较应该使用 `is` 或者 `is not`，而不是等号（==）。

  另外，不要把 `if x is not None` 写成 `if x`。比如，你想测试默认为 None 的变量或参数是否设置为其他值时，其他值可能是一种特殊类型（比如容器），这种类型在做布尔运算时被当作 false。

- 建议使用 `is not` 运算符，而不是 `not ... is`。虽然两个表达式功能相同，但前者可读性更高，应该被选用。

  > *推荐写法：*

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

  > *不推荐写法：*

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

- 用富比较实现排序操作的时候，最好实现全部六个比较运算符（ `__eq__`，`__ne__`，`__lt__`，`__le__`，`__gt__`，`__ge__` ），而不是依靠其他代码来进行特定比较。

  为最大限度的降低工作量，`functools.total_ordering()` 装饰器提供了一个工具来生成缺少的比较方法。

  PEP 207 指出 Python 实现了反射机制。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()` 操作使用了 `<` 运算符，`max()` 函数使用了 `>` 运算符。但是，最好是六个操作符都实现，以免在其他情况下出现混淆。

- 要使用定义语句，而不是利用赋值语句为某个标识符绑定 `lambda` 表达式。

  > *推荐写法：*

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

  > *不推荐写法：*

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

第一种格式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 lambda 表达式唯一优于显式 def 语句的地方，即 lambda 表达式可以嵌入到一个长语句中。

- 应从 Exception 中而不是 BaseException 中继承异常，捕获直接继承自 BaseException 的异常通常都是错误的。

  基于代码异常捕获的差异来设计异常继承的层次结构，而不是异常抛出的位置。编程的时候要以回答"出了什么问题？"为目标，而不是仅仅指出"这里出现了问题"。（可以参阅 PEP 3151 中的例子，学习内建异常层级结构的构建经验。）

  类命名规约适用于异常，但是，当异常是一种错误的话，要加上"Error"后缀。用于非本地流程控制或者其他形式的信号的非错误类异常，不需要特殊的后缀。

- 适当的使用异常链。在 Python3 中，应该使用"raise X from Y"来指示显示替换，这样不会丢失原始的回溯。

  当有意替换一个内部异常时（Python 中使用"raise X"，Python3.3+ 中使用"raise X from None"），要确保将相关细节转移到新异常中（比如，将 `KeyError` 转化成 `AttributeError` 时保留属性名称，或者将原始异常的文本嵌入到新的异常信息中）。

- 在 Python2 里抛出异常时，用 `raise ValueError('message')` 代替旧式的 `raise ValueError, 'message'` 。

  第二种格式在 Python3 中是不合法的语法。

  带括号的形式也意味着当异常参数很长或者包含格式化字符串时，你不需要使用续行符，这多亏了括号。

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

  例如：

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

  一个空的 `except:` 子句将会捕获到 SystemExit和 KeyboardInterrupt 异常，这样就很难使用 Ctrl + C 来中断程序，还会掩盖其他问题。如果你想捕获可以表示程序错误的所有异常，可以使用 `except Exception:` (空 `except:` 等同于 `except BaseException:`)。

  经验告诉我们，在以下两种情况中要限制空 `except:` 子句的使用：

  1. 异常处理程序想要打印或者记录回溯信息，至少这能使用户意识到有错误发生。
  2. 如果代码需要做一些清理工作，但是随后要用 `raise` 向上抛出异常。那么 `try...finally` 可以更好的处理这个问题。

- 当要给异常绑定一个名称时，最好使用 Python 2.6 中加入的明确的名称绑定语法：

```
try:
```

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

这是 Python3 中唯一支持的语法，并且避免了与基于逗号的旧式语法产生二义性问题。

- 在捕获操作系统错误时，首选 Python3.3 中引入的显式异常层次结构，而不是检查 `error` 值。
- 另外，对于所有的 try/except 子句，应将 try 子句限制为必要的最小的代码量。再者，可以避免掩盖问题。

> *推荐写法：*

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

> *不推荐写法：*

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

- 当某个资源仅被本地的特定代码段使用时，请使用 with 语句以确保资源使用后可以被及时可靠地清理。也可以使用 try/finally 语句。
- 只要不是获取和释放资源而是执行其他操作，上下文管理器都应该通过独立的函数或方法来调用。

> *推荐写法：*

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

> *不推荐写法*

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

第二个例子没有提供任何信息来表明：除了在事物处理之后关闭连接，__enter__ 和__exit__ 方法会做其他事情。在这种情况下，明确是很重要的。

- 坚持使用 return 语句。函数中的所有 return 语句都应该返回一个表达式或者 None。如果有 return 语句返回了一个表达式，那么，没有返回值的语句都要明确地用 `return None` 说明。如果可能的话，应该以一条清晰的 return 语句作为函数的结尾。

> *推荐写法：*

```
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。

  字符串方法通常要快很多，并且和 Unicode 字符串共享相同的 API。如果需要兼容 Python2.0 以下的版本，就需要覆盖此规则。

- 使用 `''.startswith()` 和 `''.endswith()` 而不是字符串切片操作来检查前缀和后缀。

  startswith() 和 endswith() 更简洁，且不容易出错，例如：

  > *推荐写法：*

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

  > *不推荐写法：*

  ```
  if foo[:3] == 'bar':
  ```

- 对象类型比较应该用 isinstance() 而不是直接比较。

  > *推荐写法：*

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

  > *不推荐写法：*

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

当检查一个对象是否为字符串时，注意它也可能是一个 Unicode 字符串。在 Python2 中，str 和 Unicode 有一个公共的基类即 basestring，所以你可以这样做：

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

注意，在 Python3 中，`unicode` 和 `basestring` 已经不存在了（只有 `str` ）。并且 `bytes` 对象不再是一种字符串（反而是一种整数序列）。

- 对于序列（字符串、列表、元组）来说，空序列的布尔值为 False。

  > *推荐写法：*

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

  > *不推荐写法：*

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

- 书写字符串字面值时，不要依赖尾随的空格。这样的尾随空格在视觉上难以区分，而且一些编辑器会去掉他们（或者，更进一步来说，reindent.py 就会这么做）。
- 不要将布尔量与 True 或 False 用 `==` 做比较。

  > *推荐写法：*

  ```
  if greeting:
  ```

  > *不推荐写法*

  ```
  if greeting == True:
  ```

  > *错误写法：*

  ```
  if greeting is True:
  ```

(1)函数注解

随着 PEP 484 被正式接受，函数注解的样式规则正在发生变化。

- 为了向前兼容，Python3 代码中的函数注解最好使用 PEP 484 语法。（上一节中有关于注释格式化的一些建议。）
- 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，评估添加这些注解是否容易，并观察其存在是否增加了代码的可读性。
- Python 标准库要慎用这样的注解，但是，允许用于新编写的代码或者大型的重构。
- 如果想要尝试函数注解的不同的使用方法，建议在文件顶部加入以下形式的注释：

  ```
  # 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法减少类型检查器报错。）
- 像代码检查工具一样，类型检查器也是可选的独立工具。默认情况下，Python 解释器不会因为类型检查而发出任何消息，也不会根据注解来修改行为。
- 不想使用类型检查的用户可以随意忽略他们但是，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，与相应的 .py 文件相比，类型检查器会优先读取这类文件。存根文件可以与库一起发布，也可以在作者许可的情况下通过 typeshed repo 单独发布。
- 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。可参阅 PEP 484 的相关部分。

（2）变量注解

PEP 526 引入了变量注解。他们的风格建议与上面介绍的函数注解类似：

- 模块级变量，类和实例变量以及局部变量的注解在冒号后面应该有一个空格。
- 冒号前面不应该有空格。
- 如果赋值号有右值，那么等号两边都应该有一个空格。

  > *推荐写法：*

  ```
  code: int

  class Point:
      coords: Tuple[int, int]
      label: str = ''
  ```

  > *不推荐写法：*

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

  class Test:
      result: int=0  # No spaces around equality sign
  ```
- 虽然 PEP 526 已被 Python3.6 所接受，但是对于所有 Python 版本中的存根文件，首选变量注解语法（详见 PEP 484）。

脚注

悬挂缩进是一种排版样式，除第一行外，其段落中的所有行都要缩进。在 `Python` 文本中，这一术语描述了这样一种样式：对于括号括起来的语句，该行的最后一个非空白字符是左括号，后续行都要缩进，直到右括号。

10、参考文档

编号	内容
[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

11、版权

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