

# PEP8 代码风格指南

---

## 一、实验介绍

---

### 1.1 实验内容

编程语言不是艺术，而是工作或者说是工具，所以整理并遵循一套编码规范是十分必要的。

这篇文章原文实际上来自于这里：<https://www.python.org/dev/peps/pep-0008/> (<https://www.python.org/dev/peps/pep-0008/>)

### 1.2 知识点

- 代码排版
- 字符串引号
- 表达式和语句中的空格
- 注释
- 版本注记
- 命名约定
- 公共和内部接口
- 程序编写建议

### 1.3 实验环境

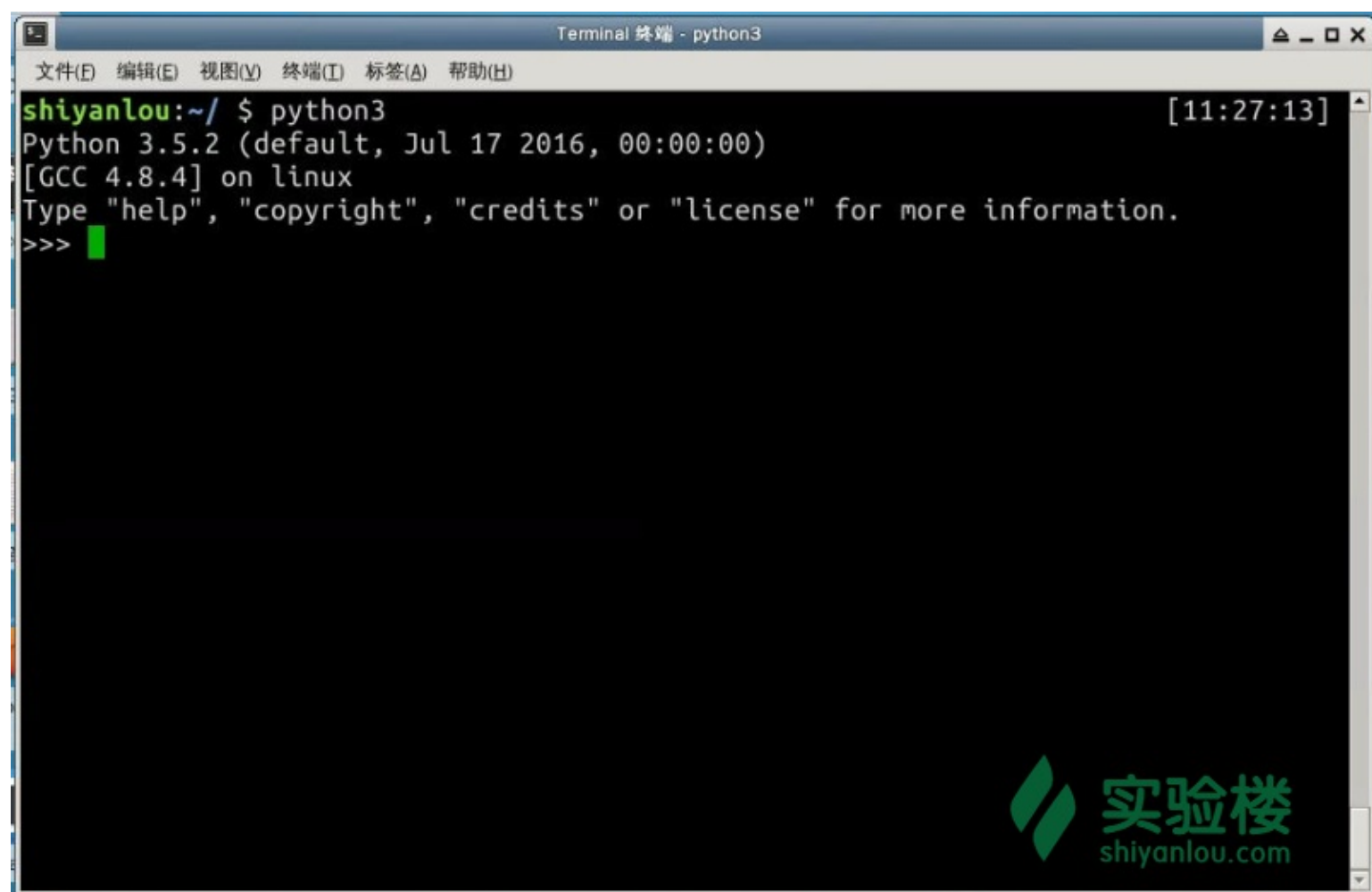
- python3.5
- Xfce终端
- Vim

## 1.4 适合人群

本课程属于初级级别课程，不仅适用于那些有其它语言基础的同学，对没有编程经验的同学也非常友好

## 二、实验步骤

建议在实验楼中打开 Python 解释器或者 vim 自己照着做一下，或者看看以前自己写的代码

A screenshot of a terminal window titled "Terminal 终端 - python3". The window has a menu bar with "文件(F)", "编辑(E)", "视图(V)", "终端(T)", "标签(A)", and "帮助(H)". The terminal content shows the command "python3" being executed in a shell at the "shiyanolou" prompt. The output displays the Python version "Python 3.5.2 (default, Jul 17 2016, 00:00:00)", the compiler "[GCC 4.8.4] on linux", and a prompt to type "help", "copyright", "credits", or "license" for more information. The prompt ">>>" is followed by a green cursor. In the bottom right corner, there is a green logo and the text "实验楼 shiyanlou.com".

```
Terminal 终端 - python3
文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)
shiyanolou:~/ $ python3 [11:27:13]
Python 3.5.2 (default, Jul 17 2016, 00:00:00)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



## 2.1 介绍

这份文档给出的代码约定适用于主要的 Python 发行版所有标准库中的 Python 代码。请参阅相似的 PEP 信息，其用于描述实现 Python 的 C 代码规范[1]。

这份文档和 PEP 257 (<https://www.python.org/dev/peps/pep-0257>)(文档字符串约定) 改编自 Guido 的 Python 风格指南原文，从 Barry 的风格指南里添加了一些东西[2]。

随着时间的推移，这份额外约定的风格指南已经被认可了，过去的约定由于语言自身的发展被淘汰了。

许多项目有它们自己的编码风格指南。如果有冲突，优先考虑项目规定的编码指南。

## 2.2 愚蠢的一致性就像没脑子的妖怪

Guido 的一个主要见解是读代码多过写代码。这里提供指南的意图是强调代码可读性的重要性，并且使大多数 Python 代码保持一致性。如 PEP 20

(<https://www.python.org/dev/peps/pep-0020>) 所述，“Readability counts”。

风格指南是关于一致性的。风格一致对于本指南来说是重要的，对于一个项目来说是更重要的，对于一个模块或者方法来说是最重要的。

但是最最重要的是：知道什么时候应该破例—有时候这份风格指南就是不适用。有疑问时，用你最好的判断力，对比其它的例子来确定这是不是最好的情况，并且不耻下问。

特别说明：不要为了遵守这份风格指南而破坏代码的向后兼容性。

这里有一些好的理由去忽略某个风格指南：

1. 当应用风格指南的时候使代码更难读了，对于严格依循风格指南的约定去读代码的人也是不应该的。
2. 为了保持和风格指南的一致性同时也打破了现有代码的一致性（可能是历史原因）—虽然这这也是一个整理混乱代码的机会（现实中的 XP 风格）。
3. 因为问题代码的历史比较久远，修改代码就没有必要性了。
4. 当代码需要与旧版本的 Python 保持兼容，而旧版 Python 又不支持风格指南中提到的特性的时候。

## 2.3 代码排版

### 2.3.1 缩进

每层缩进使用4个空格。

续行要么与圆括号、中括号、花括号这样的被包裹元素保持垂直对齐，要么放在 Python 的隐线（注：应该是相对于def的内部块）内部，或者使用悬挂缩进。使用悬挂缩进的注意事项：第一行不能有参数，用进一步的缩进来把其他行区分开。

好的：

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

# More indentation included to distinguish this 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)
```

不好的：

```
# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
                          var_three, var_four)

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

4空格规则是可选的：

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

当 if 语句的条件部分足够长，需要将它写入到多个行，值得注意的是两个连在一起的关键字（i.e. if），添加一个空格，给后续的多行条件添加一个左括号形成自然地4空格缩进。如果和嵌套在 if 语句内的缩进代码块产生了视觉冲突，也应该被自然缩进4个空格。这份增强建议书对于怎样（或是否）把条件行和 if 语句的缩进块在视觉上区分开来是没有明确规定的。可接受的情况包括，但不限于：

```
# 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()
```

在多行结构中的右圆括号、右中括号、右大括号应该放在最后一行的第一个非空白字符的正下方，如下所示：

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

或者放在多行结构的起始行的第一个字符正下方，如下：

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

## 2.3.2 制表符还是空格？

空格是首选的缩进方法。

制表符 ( Tab ) 应该被用在那些以前就使用了制表符缩进的地方。

Python 3 不允许混合使用制表符和空格来缩进代码。

混合使用制表符和空格缩进的 Python 2 代码应该改为只使用空格。

当使用 `-t` 选项来调用 Python 2 的命令行解释器的时候，会对混合使用制表符和空格的代码发出警告。当使用 `-tt` 选项的时候，这些警告会变成错误。这些选项是强烈推荐的！

### 2.3.3 每行最大长度

限制每行的最大长度为79个字符。

对于那些约束很少的文本结构（文档字符串或注释）的长块，应该限制每行长度为72个字符。

限制编辑窗口的宽度使并排打开两个窗口成为可能，使用通过代码审查工具时，也能很好的通过相邻列展现不同代码版本。

一些工具的默认换行设置打乱了代码的可视结构，使其更难理解。限制编辑器窗口宽为80来避免自动换行，即使有些编辑工具在换行的时候会在最后一列放一个标识符。一些基于 Web 的工具可能根本就不提供动态换行。

一些团队更倾向于长的代码行。对于达成了一致意见来统一代码的团队而言，把行提升到80~100的长度是可接受的（实际最大长度为99个字符），注释和文档字符串的长度还是建议在72个字符内。

Python 标准库是非常专业的，限制最大代码长度为79个字符（注释和文档字符串最大长度为72个字符）。

首选的换行方式是在括号（小中大）内隐式换行（非续行符 `\` ）。长行应该在括号表达式的包裹下换行。这比反斜杠作为续行符更好。

反斜杠有时仍然适用。例如，多个很长的 `with` 语句不能使用隐式续行，因此反斜杠是可接受的。

```
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())
```

(见前面关于多行 if 语句 (<https://www.python.org/dev/peps/pep-0008/#multiline-if-statements>)的讨论来进一步思考这种多行 with 语句该如何缩进)

另一种使用反斜杠续行的案例是 assert 语句。

确保续行的缩进是恰到好处的。遇到二元操作符，首选的断行位置是操作符的后面而不是前面。这有一些例子：

```
class Rectangle(Blob):

    def __init__(self, width, height,
                  color='black', emphasis=None, highlight=0):
        if (width == 0 and height == 0 and
            color == 'red' and emphasis == 'strong' or
            highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                              (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

## 2.3.4 空行

顶级函数和类定义上下使用两个空行分隔。

类内的方法定义使用一个空行分隔。

可以使用额外的空行（有节制的）来分隔相关联的函数组。在一系列相关联的单行代码中空行可以省略（e.g. 一组虚拟的实现）。

在函数中使用空白行（有节制的）来表明逻辑部分。



Python 接受使用换页符 ( i.e. Ctrl+L ) 作为空格；许多工具都把 Ctrl+L 作为分页符，因此你可以用它们把你的文件中相似的章节分页。注意，一些编辑器和基于 Web 的代码查看工具可能不把 Ctrl+L 看做分页符，而是在这个位置放一个其它的符号。

## 2.3.5 源文件编码

在核心 Python 发布版中的代码应该总是使用 UTF-8 编码（或者在 Python 2 中使用 ASCII ）。

使用 ASCII (Python 2)或 UTF-8 (Python 3)的文件不需要有编码声明（注：它们是默认的）。

在标准库中，非缺省的编码应该仅仅用于测试目的，或者注释或文档字符串中的作者名包含非 ASCII 码字符；否则，优先使用 `\x`、`\u`、`\U` 或者 `\N` 来转义字符串中的非 ASCII 数据。

对于 Python 3.0 和之后的版本，以下是有关标准库的政策（见 PEP 3131 (<https://www.python.org/dev/peps/pep-3131>)）：所有 Python 标准库中的标识符必须使用只含 ASCII 的标识，并且应该使用英语单词只要可行（在多数情况下，缩略语和技术术语哪个不是英语）。此外，字符串和注释也必须是 ASCII。仅有的例外是：(a)测试用例测试非 ASCII 特性时，(b)作者名。作者的名字不是基于拉丁字母的必须提供他们名字的拉丁字母音译。

面向全球用户的开源项目，鼓励采取相似的政策。

## 2.3.6. 导入包

- `import` 不同的模块应该独立一行，如：

好的:

```
import os
import sys
```

不好的:

```
import sys, os
```

这样也是可行的：

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

- `import` 语句应该总是放在文件的顶部，在模块注释和文档字符串之下，在模块全局变量和常量之前。

`import` 语句分组顺序如下：

- i. 导入标准库模块
- ii. 导入相关第三方库模块
- iii. 导入当前应用程序/库模块

每组之间应该用空行分开。

然后用 `__all__` 声明本文件内的模块。

- 绝对导入是推荐的，它们通常是更可读的，并且在错误的包系统配置（如一个目录包含一个以 `os.path` 结尾的包）下有良好的行为倾向（至少有更清晰的错误消息）：

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

当然，相对于绝对导入，相对导入是个可选替代，特别是处理复杂的包结构时，绝对导入会有不必要的冗余：

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

标准库代码应该避免复杂的包结构，并且永远使用绝对导入。

应该从不使用隐式的相对导入，而且在 Python 3 中已经被移除。

- 从一个包含类的模块导入类时，这样写通常是可行的：

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

如果上面的方式会本地导致命名冲突，则这样写：

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

以 `myclass.MyClass` 和 `foo.bar.yourclass.YourClass` 这样的方式使用。

- 应该避免通配符导入（`from import *`），这会使名称空间里存在的名称变得不清晰，迷惑读者和自动化工具。这里有一个可辩护的通配符导入用例，，重新发布一个内部接口作为公共 API 的一部分（例如，使用纯 Python 实现一个可选的加速器模块的接口，但并不能预知这些定义会被覆盖）。

当以这种方式重新发布名称时，下面关于公共和内部接口的指南仍然适用。

## 2.4 字符串引号

在 Python 里面，单引号字符串和双引号字符串是相同的。这份指南对这个不会有所建议。选择一种方式并坚持使用。一个字符串同时包含单引号和双引号字符时，用另外一种来包裹字符串，而不是使用反斜杠来转义，以提高可读性。

对于三引号字符串，总是使用双引号字符来保持与文档字符串约定的一致性（PEP 257 (<https://www.python.org/dev/peps/pep-0257>)）。

## 2.5 表达式和语句中的空格

### 2.5.1 不能忍受的情况

避免在下列情况中使用多余的空格：

- 与括号保持紧凑（小括号、中括号、大括号）：

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

- 与后面的逗号、分号或冒号保持紧凑：

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

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

- 切片内的冒号就像二元操作符一样，任意一侧应该被等同对待（把它当做一个极低优先级的操作）。在一个可扩展的切片中，冒号两侧必须有相同的空格数量。例外：切片参数省略时，空格也省略。

好的：

`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]`

不好的：

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

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

`ham[lower : : upper]`

`ham[ : upper]`

- 函数名与其后参数列表的左括号应该保持紧凑：

Yes: `spam(1)`

No: `spam (1)`

- 与切片或索引的左括号保持紧凑：

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

No: `dct ['key'] = lst [index]`

- 在复制操作符（或其它）的两侧保持多余一个的空格：

好的：

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

不好的：

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

## 2.5.2 其他建议

- 总是在这些二元操作符的两侧加入一个空格：赋值(=)，增量赋值(+=, -= etc.)，比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not)，布尔运算(and, or, not)。
- 在不同优先级之间，考虑在更低优先级的操作符两侧插入空格。用你自己的判断力；但不要使用超过一个空格，并且在二元操作符的两侧有相同的空格数。

好的：

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

不好的：

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

- 不要在关键值参数或默认值参数的等号两边加入空格。

好的：

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

不好的：

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

- **【注：Python 3】**带注释的函数定义中的等号两侧要各插入空格。此外，在冒号后用一个单独的空格，也要在表明函数返回值类型的 -> 左右各插入一个空格。

好的：

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

不好的：

```
def munge(input: AnyStr=None):  
def munge(input:AnyStr):  
def munge(input: AnyStr)->PosInt:
```

- 打消使用复合语句（多条语句在同一行）的念头。

好的：

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

宁可不：

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

- 有时候把 if/for/while 和一个小的主体放在同一行也是可行的，千万不要在有多条语句的情况下这样做。此外，还要避免折叠，例如长行。

宁可不：

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

绝对不：

```
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()
```

## 2.6 注释

与代码相矛盾的注释不如没有。注释总是随着代码的变更而更新。

注释应该是完整的句子。如果注释是一个短语或语句，第一个单词应该大写，除非是一个开头是小写的标识符（从不改变标识符的大小写）。

如果注释很短，末尾的句点可以省略。块注释通常由一个或多个有完整句子的段落组成，并且每个句子应该由句点结束。

你应该在一个句子的句点后面用两个空格。

写英语时，遵循《Strunk and White》（注：《英文写作指南》，参考维基百科 ([https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=13&cad=rja&uact=8&ved=0CJgBEJoTKAAwDA&url=http%3A%2F%2Fzh.wikipedia.org%2Fzh-cn%2F%25E8%258B%25B1%25E6%2596%2587%25E5%2586%2599%25E4%25BD%259C%25E6%258C%2587%25E5%258D%2597&ei=WGU7VaiiJITxoATO4YCwAw&usg=AFQjCNE3ld8HgP5OVy1WUoMQ\\_vVsUT8-Ng&sig2=vcim1a38TCK3svBegulR4w](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=13&cad=rja&uact=8&ved=0CJgBEJoTKAAwDA&url=http%3A%2F%2Fzh.wikipedia.org%2Fzh-cn%2F%25E8%258B%25B1%25E6%2596%2587%25E5%2586%2599%25E4%25BD%259C%25E6%258C%2587%25E5%258D%2597&ei=WGU7VaiiJITxoATO4YCwAw&usg=AFQjCNE3ld8HgP5OVy1WUoMQ_vVsUT8-Ng&sig2=vcim1a38TCK3svBegulR4w)) ）。

来自非英语国家的程序员：请用英语写注释，除非你120%确定你的代码永远不会被那些不说你的语言的人阅读。

## 2.6.1 块注释

块注释通常用来说明跟随在其后的代码，应该与那些代码有相同的缩进层次。块注释每一行以 # 起头，并且 # 后要跟一个空格（除非是注释内的缩进文本）。

## 2.6.2 行内注释

有节制的使用行内注释。

一个行内注释与语句在同一行。行内注释应该至少与语句相隔两个空格。以 # 打头，# 后接一个空格。

无谓的行内注释如果状态明显，会转移注意力。不要这样做：

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

但有的时候，这样是有用的：

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

## 2.6.3 文档字符串

编写良好的文档字符串（a.k.a “docstring”）的约定常驻在 PEP 257 (<https://www.python.org/dev/peps/pep-0257>)



- 为所有的公共模块、函数、类和方法编写文档字符串。对于非公共的方法，文档字符串是不必要的，但是也应该有注释来说明代码是干什么的。这个注释应该放在方法声明的下面。
- PEP 257 (<https://www.python.org/dev/peps/pep-0257>) 描述了良好的文档字符串的约定。注意，文档字符串的结尾 `"""` 应该放在单独的一行，例如：

```
"""Return a foobang

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

- 对于单行的文档字符串，把结尾 `"""` 放在同一行。

## 2.7 版本注记

如果必须要 Subversion，CVS 或 RCS 标记在你的源文件里，像这样做：

```
__version__ = "$Revision$"  
# $Source$
```

这几行应该在模块的文档字符串后面，其它代码的前面，上下由一个空行分隔。

## 2.8 命名约定

Python 库的命名规则有点混乱，因此我们永远也不会使其完全一致的 – 不过，这里有一些当前推荐的命名标准。新的模块和包（包括第三方框架）应该按照这些标准来命名，但是已存在库有不同的风格，内部一致性是首选。

### 2.8.1 覆盖原则

API 里对用户可见的公共部分应该遵循约定，反映的是使用而不是实现。

### 2.8.2 规定：命名约定

有许多不同的命名风格。这有助于识别正在使用的命名风格，独立于它们的用途。

下面的命名风格通常是有区别的：

- b (一个小写字母)
- B (一个大写字母)
- lowercase
- lower\_case\_with\_underscores
- UPPERCASE
- UPPER\_CASE\_WITH\_UNDERSCORES
- CapitalizedWords (又叫 CapWords，或者 CamelCase(骆驼命名法) – 如此命名因为字母看起来崎岖不平[3]。有时候也叫 StudlyCaps。

注意：在 CapWords 使用缩略语时，所有缩略语的首字母都要大写。因此 `HTTPErverError` 比 `HttpServerError` 要好。

- mixedCase (和上面不同的是首字母小写)
- Capitalized\_Words\_With\_Underscores (丑陋无比！)

也有种风格用独一无二的短前缀来将相似的命名分组。在 Python 里用的不是很多，但是为了完整性被提及。例如，`os.stat()` 函数返回一个元组，通常有像 `st_mode`，`st_size`，`st_mtime` 等名字。（强调与 POSIX 系统调用的字段结构一致，有助于程序员对此更熟悉）

X11 库的所有公共函数都用 X 打头。在 Python 中这种风格被认为是不重要的，因为属性和方法名的前缀是一个对象，函数名的前缀为一个模块名。

此外，下面的特许形式用一个前导或尾随的下划线进行识别（这些通常可以和任何形式的命名约定组合）：

- `_single_leading_underscore`：仅内部使用的标识，如 `from M import *` 不会导入像这样一下划线开头的对象。
- `single_trailing_underscore`：通常是为了避免与 Python 规定的关键字冲突，如 `Tkinter.Toplevel(master, class_='ClassName')`。
- `double_leading_underscore`：命名一个类属性，调用的时候名字会改变

(在类 **FooBar** 中, ``boo`` 变成了 `_FooBar__boo``; 见下)。

- `double_leading_and_trailing_underscore` : “魔术”对象或属性, 活在用户控制的命名空间里。例如, `__init__`, `__import__` 和 `__file__`。永远不要像这种方式命名; 只把它们作为记录。

## 2.8.3 规定：命名约定

### 2.8.3.1 应该避免的名字

永远不要使用单个字符 `l` (小写字母 `el`), `0` (大写字母 `oh`), 或 `I` (大写字母 `eye`) 作为变量名。

在一些字体中, 这些字符是无法和数字 `1` 和 `0` 区分开的。试图使用 `l` 时用 `L` 代替。

### 2.8.3.2 包和模块名

模块名应该短, 且全小写。如果能改善可读性, 可以使用下划线。Python 的包名也应该短, 全部小写, 但是不推荐使用下划线。

因为模块名就是文件名, 而一些文件系统是大小写不敏感的, 并且自动截断长文件名, 所以给模块名取一个短小的名字是非常重要的 – 在 Unix 上这不是问题, 但是把代码放到老版本的 Mac, Windows, 或者 DOS 上就可能变成一个问题了。

用 C/C++ 给 Python 写一个高性能的扩展(e.g. more object oriented)接口的时候, C/C++ 模块名应该有一个前导下划线。

### 2.8.3.3 类名

类名通常使用 CapWords 约定。

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

注意和内建名称的区分开：大多数内建名称是一个单独的单词（或两个单词一起），CapWords 约定只被用在异常名和内建常量上。

### 2.8.3.4 异常名

因为异常应该是类，所以类名约定在这里适用。但是，你应该用 `Error` 作为你的异常名的后缀（异常实际上是一个错误）。

### 2.8.3.5 全局变量名

（我们希望这些变量仅仅在一个模块内部使用）这个约定有关诸如此类的变量。

若被设计的模块可以通过 `from M import *` 来使用，它应该使用 `__all__` 机制来表明那些可以可导出的全局变量，或者使用下划线前缀的全局变量表明其是模块私有的。

### 2.8.3.6 函数名

函数名应该是小写的，有必要的话用下划线来分隔单词提高可读性。

混合大小写仅仅在上下文都是这种风格的情况下允许存在（如 `thread.py`），这是为了维持向后兼容性。

### 2.8.3.7 函数和方法参数

总是使用 `self` 作为实例方法的第一个参数。

总是使用 `cls` 作为类方法的第一个参数。

如果函数参数与保留关键字冲突，通常最好在参数后面添加一个尾随的下划线，而不是使用缩写或胡乱拆减。因此 `class_` 比 `clss` 要好。（或许避免冲突更好的方式是使用近义词）

### 2.8.3.8 方法名和实例变量

用函数名的命名规则：全部小写，用下划线分隔单词提高可读性。

用一个且有一个前导的下划线来表明非公有的方法和实例变量。

为了避免与子类变量或方法的命名冲突，用两个前导下划线来调用 Python 的命名改编规则。

Python 命名改编通过添加一个类名：如果类 `Foo` 有一个属性叫 `__a`，它不能被这样 `Foo.__a` 访问（执着的人可以通过这样 `Foo._Foo__a` 来访问）通常，双前导的下划线应该仅仅用来避免与其子类属性的命名冲突。

注意：这里有一些争议有关 `__names` 的使用（见下文）。

### 2.8.3.9 常量

常量通常是模块级的定义，全部大写，单词之间以下划线分隔。例如 `MAX_OVERFLOW` 和 `TOTAL`。

### 2.8.3.10 继承的设计

总是决定一个类的方法和变量（属性）是应该公有还是非公有。如果有疑问，选择非公有；相比把共有属性变非公有，非公有属性变公有会容易得多。

公有属性是你期望给那些与你的类无关的客户端使用的，你应该保证不会出现不向后兼容的改变。非公有的属性是你不打算给其它第三方使用的；你不需要保证非公有的属性不会改变甚至被移除也是可以的。

我们这里不适用“私有”这个术语，因为在 Python 里没有真正的私有属性（一般没有不必要的工作量）。

另一种属性的分类是“子类 API”的一部分（通常在其它语言里叫做“Protected”）。一些类被设计成被继承的，要么扩展要么修改类的某方面行为。设计这样一个类的时候，务必做出明确的决定，哪些是公有的，其将会成为子类 API 的一部分，哪些仅仅是用于你的基类的。

处于这种考虑，给出 Pythonic 的指南：

- 共有属性不应该有前导下划线。
- 如果你的公有属性与保留关键字发生冲突，在你的属性名后面添加一个尾随的下划线。这比使用缩写或胡乱拆减要好。（尽管这条规则，已知某个变量或参数可能是一个类情况下，`cls` 是首选的命名，特别是作为类方法的第一个参数）

注意一：见上面推荐的类方法参数命名方式。

- 对于简单的公有数据属性，最好的方式是暴露属性名，不要使用复杂的访问属性/修改属性的方法。记住，Python 提供了捷径去提升特性，如果你发现简单的数据属性需要增加功能行为。在这种情况下，使用 properties 把功能实现隐藏在简单的数据属性访问语法下面。

注意一：properties 仅仅在新式类下工作。      注意二：尽量保持功能行为无边际效应，然而如缓存有边际效应也是好的。      注意三：避免为计算开销大的操作使用 properties；属性标记使调用者相信这样来访问（相对来说）是开销很低的。

- 如果你的类是为了被继承，你有一不想让子类使用的属性，给属性命名时考虑给它们加上双前导下划线，不要加尾随下划线。这会调用 Python 的名称重整算法，把类名加在属性名前面。避免了命名冲突，当子类不小心命名了和父类属性相同名称的时候。

注意一：注意只是用了简单的类名来重整名字，因此如果子类 and 父类同名的时候，你仍然有能力避免冲突。

注意二：命名重整有确定的用途，例如调试和 `__getattr__()`，就不太方便。命名重整算法是有据可查的，易于手动执行。

注意三：不是每个人都喜欢命名重整。尽量平衡名称的命名冲突与面向高级调用者的潜在用途。

## 2.9 公共和内部接口

保证所有公有接口的向后兼容性。用户能清晰的区分公有和内部接口是重要的。

文档化的接口考虑公有，除非文档明确的说明它们是暂时的，或者内部接口不保证其的向后兼容性。所有的非文档化的应该被假设为非公开的。

为了更好的支持内省，模块应该用 `__all__` 属性来明确规定公有 API 的名字。设置 `__all__` 为空 list 表明模块没有公有 API。

甚至与 `__all__` 设置相当，内部接口（包、模块、类、函数、属性或者其它的名字）应该有一个前导的下划线前缀。

被认为是内部的接口，其包含的任何名称空间（包、模块或类）也被认为是内部的。

导入的名称应始终视作一个实现细节。其它模块不能依赖间接访问这些导入的名字，除非它们是包含模块的 API 明确记载的一部分，例如 `os.path` 或一个包的 `__init__` 模块暴露了来自子模块的功能。

## 2.10 程序编写建议

- 代码的编写方式不能对其它 Python 的实现（PyPy、Jython、IronPython、Cython、Psyco，诸如此类的）不利。

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。

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

当你的真正用意是 `if x is not None` 的时候，当心 `if x` 这样的写法 – 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，其它值可能是那些布尔值为 `false` 的类型（如空容器）。

- 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。

好的:

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

不好的:

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

- 用富比较实现排序操作的时候，实现所有六个比较操作符（ `__eq__` 、 `__ne__` 、 `__lt__` , `__le__` , `__gt__` , `__ge__` ）是更好的，而不是依赖其它仅仅运用一个特定比较的代码

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 (<https://www.python.org/dev/peps/pep-0207>) 说明了 Python 假定的所有反射规则。因此，解释器可能交换  $y > x$  与  $x < y$  ,  $y \geq x$  与  $x \leq y$  , 也可能交换  $x == y$  和  $x != y$  。 `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，以便不会在其它上下文中有疑惑。

- 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。

好的:

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

不好的:

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

第一种形式意味着函数对象的名字是 `f` 而不是 `'` 的。通常这对异常追踪和字符串表述是更有用的。使用赋值语句消除的唯一好处， `lambda` 表达式可以提供 一个显示的 `def` 语句不能提供的，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` 。直接继承自 `BaseException` 是为 `Exception` 保留的，如果从 `BaseException` 继承，捕获到的错误总是错的。

设计异常结构层次，应基于那些可能出现异常的代码，而不是在出现异常后的。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 (<https://www.python.org/dev/peps/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，`raise X from Y` 用于表明明确的替代者，不丢失原有的回溯信息。

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`），确保相关的细节全部转移给了新异常（例如，把 `KeyError` 变成 `AttributeError` 时保留属性名，或者把原始异常的错误信息嵌在新异常里）。

- 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 .. fin`

ally 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，而不是自省的 `errno` 值。
- 此外，对于所有的 `try/except` 语句来说，限制 `try` 里面有且仅有绝对必要的代码。在强调一次，这能避免屏蔽错误。

好的：

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

不好的：

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

- 当资源是本地的特定代码段，用 `with` 语句确保其在使用后被立即干净的清除了，`try/finally` 也是接受的。

- 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：

好的：

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

不好的：

```
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）。

好的：

```
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)
```

不好的：

```
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。

字符串方法总是更快，与 unicode 字符串共享 API。如果需要向后兼容性覆盖这个规则，需要 Python 2.0 以上的版本。

- 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：

```
Yes: if foo.startswith('bar'):
No:  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。

```
Yes: if isinstance(obj, int):
No:  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`：

好的：

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

不好的：

```
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（or more recently, reindent.py）会将其裁剪掉。
- 不要用 `==` 比较 `True` 和 `False`。

```
Yes:    if greeting:
No:     if greeting == True:
Worse:  if greeting is True:
```

- Python 标准库将不再使用函数标注，以至于给特殊的标注风格给一个过早的承若。代替的，这些标注是留给用户去发现和体验的有用的标注风格。

建议第三方实验的标注用相关的修饰符指示标注应该如何被解释。

早期的核心开发者尝试用函数标注显示不一致、特别的标注风格。例如：

- `[str]` 是很含糊的，它可能代表一个包含字符串的列表，也可能代表一个为字符串或为空的值。
- `open(file:(str,bytes))` 可能用来表示 `file` 的值可以是一个 `str` 或者 `bytes`，也可能用来表示 `file` 的值是一个包含 `str` 和 `bytes` 的二元组。
- 标注 `seek(whence:int)` 体现了一个过于明确又不够明确的混合体：`int` 太严格了（有 `__index__` 的应该被允许），又不够严格（只有 `0,1,2` 是被允许的）。同样的，标注 `write(b: byte)` 太严格了（任何支持缓存协议的都应该被允许）。
- 像 `read1(n: int=None)` 这样的标注自我矛盾，因为 `None` 不是 `int`。像 `source_path(self, fullname:str) -> object` 标注是迷惑人的，返回值到底是应该什么类型？
- 除了上面之外，在具体类型和抽象类型的使用上是不一致的：`int` 对 `int`

egral ( 整数 ) , set/fronzenset 对 MutableSet/Set 。

- 不正确的抽象基类标注规格。例如，集合之间的操作需要另一个对象是集合的实例，而不只是一个可迭代序列。
- 另一个问题是，标注成为了规范的一部分，但却没有经受过考验。
- 在大多数情况下，文档字符串已经包括了类型规范，比函数标注更清晰。在其余的情况下，一旦标注被移除，文档字符串应该被完善。
- 观察到的函数标注太标新立异了，相关的系统不能一致的处理自动类型检查和参数验证。离开这些标注的代码以后很难做出更改，使自动化工具可以支持。

## 三、总结

---

即使内容有点多，但每一个 Python 开发者都应该尽量遵守 PEP8

## 四、参考链接

---

### 4.1 参考文献

[1] : PEP 7 (<https://www.python.org/dev/peps/pep-0007>), Style Guide for C Code, van Rossum

[2] : Barry's GNU Mailman style guide

<http://barry.warsaw.us/software/STYLEGUIDE.txt>

(<http://barry.warsaw.us/software/STYLEGUIDE.txt>)

[3] : <http://www.wikipedia.com/wiki/CamelCase>

(<http://www.wikipedia.com/wiki/CamelCase>)

### 4.2 版权

This document has been placed in the public domain.

Source: <https://hg.python.org/peps/file/tip/pep-0008.txt>  
(<https://hg.python.org/peps/file/tip/pep-0008.txt>)

*\*本课程内容，由作者授权实验楼发布，未经允许，禁止转载、下载及非法传播。*

上一节：挑战：类和Collection模块 (</courses/596/labs/2775/document>)

下一节：迭代器、生成器、装饰器 (</courses/596/labs/2050/document>)