# Google Python 编程风格指南

原文出处：http://code.google.com/p/zh-google-styleguide/
本文整理：li3p AT yahoo.cn

## 目录

# Python 风格规范

## 分号

- **不要在行尾加分号，也不要用分号将两条命令放在同一行.**

## 行长度

- **每行不超过 80 个字符**

  例外：如果使用 Python 2.4 或更早的版本，导入模块的行可能多于 80 个字符.

  Python 会将圆括号，中括号和花括号中的行隐式的连接起来，你可以利用这个特点. 如果需要，你可以在表达式外围增加一对额外的圆括号.

Yes:
```
foo_bar(self, width, height, color='black', design=None, x='foo',
        emphasis=None, highlight=0)

    if (width == 0 and height == 0 and
        color == 'red' and emphasis == 'strong'):
```

如果一个文本字符串在一行放不下，可以使用圆括号来实现隐式行连接:

```
x = ('This will build a very long long '
    'long long long long long long string')
```

注意上面例子中的元素缩进；你可以在本文的 *缩进* 部分找到解释.

## 括号

- **宁缺毋滥的使用括号**

除非是用于实现行连接，否则不要在返回语句或条件语句中使用括号. 不过在元组两边使用括号是可以的.

Yes:
```
if foo:
    bar()
while x:
    x = bar()
if x and y:
    bar()
if not x:
```

```
        bar()
    return foo
    for (x, y) in dict.items(): ...
```

No:
```
    if (x):
        bar()
    if not(x):
        bar()
    return (foo)
```

## 缩进

- **用 4 个空格来缩进代码**

绝对不要用 tab，也不要 tab 和空格混用．对于行连接的情况，你应该要么垂直对齐换行的元素(见 *行长度* 部分的示例)，或者使用 4 空格的悬挂式缩进(这时第一行不应该有参数)：

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

    # 4-space hanging indent; nothing on first line
    foo = long_function_name(
        var_one, var_two, var_three,
        var_four)
```

No:
```
    # Stuff on first line forbidden
    foo = long_function_name(var_one, var_two,
        var_three, var_four)

    # 2-space hanging indent forbidden
    foo = long_function_name(
      var_one, var_two, var_three,
      var_four)
```

## 空行

- **顶级定义之间空两行，方法定义之间空一行**

顶级定义之间空两行，比如函数或者类定义．方法定义，类定义与第一个方法之间，都应该空一行．函数或方法中，某些地方要是你觉得合适，就空一行．

# 空格

- **按照标准的排版规范来使用标点两边的空格**

1. 括号内不要有空格.

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

2. 不要在逗号，分号，冒号前面加空格，但应该在它们后面加(除了在行尾).

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

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

3. 参数列表，索引或切片的左括号前不应加空格.

```
Yes: spam(1)
Yes: spam (1)
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

4. 在二元操作符两边都加上一个空格，比如赋值(=)，比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not)，布尔(and, or, not)．至于算术操作符两边的空格该如何使用，需要你自己好好判断．不过两侧务必要保持一致.

```
Yes: x == 1
No:  x<1
```

5. 当’=’用于指示关键字参数或默认参数值时，不要在其两侧使用空格.

```
Yes: def complex(real, imag=0.0): return magic(r=real, i=imag)
No:  def complex(real, imag = 0.0): return magic(r = real, i = imag)
```

6. 不要用空格来垂直对齐多行间的标记，因为这会成为维护的负担(适用于:, #, =等):

```
Yes:
  foo = 1000  # comment
  long_name = 2  # comment that should not be aligned

  dictionary = {
```

```
    "foo": 1,
    "long_name": 2,
    }
```

No:

```
  foo        = 1000  # comment
  long_name  = 2     # comment that should not be aligned

  dictionary = {
     "foo"     : 1,
     "long_name": 2,
     }
```

# Python 解释器

- **每个模块都应该以#!/usr/bin/env python<version>开头**

模块应该以一个构造行开始，以指定执行这个程序用到的 Python 解释器:

```
#!/usr/bin/env python2.4
```

总是使用最特化的版本，例如，使用/usr/bin/python2.4，而不是 /usr/bin/python2. 这样，当升级到不同的 Python 版本时，能轻松找到依赖关系，同时也避免了使用时的迷惑. 例如，/usr/bin/python2 是表示/usr/bin/python2.0.1 还是/usr/bin/python2.3.0?

# 注释

- **确保对模块，函数，方法和行内注释使用正确的风格**

**文档字符串**

Python 有一种独一无二的的注释方式：使用文档字符串. 文档字符串是包，模块，类或函数里的第一个语句. 这些字符串可以通过对象的__doc__成员被自动提取，并且被 pydoc 所用. (你可以在你的模块上运行 pydoc 试一把，看看它长什么样). 我们对文档字符串的惯例是使用三重双引号. 一个文档字符串应该这样组织：首先是一行以句号，问号或惊叹号结尾的概述. 接着是一个空行. 接着是文档字符串剩下的部分，它应该与文档字符串的第一行的第一个引号对齐. 下面有更多文档字符串的格式化规范.

**模块**

每个文件应该包含下列项，依次是:

1. 版权声明(例如, Copyright 2008 Google Inc.)

2. 一个许可样板. 根据项目使用的许可(例如, Apache 2.0, BSD, LGPL, GPL), 选择合适的样板
3. 作者声明, 标识文件的原作者.

**函数和方法**

如果不是既显然又简短, 任何函数或方法都需要一个文档字符串. 而且, 任何外部可访问的函数或方法, 不管多短多简单, 都需要文档字符串. 文档字符串应该包含函数做什么, 以及输入和输出的详细描述. 通常, 不应该描述”怎么做”, 除非是一些复杂的算法. 对于技巧性的代码, 块注释或者行内注释是最重要的. 文档字符串应该提供足够的信息, 当别人编写代码调用该函数时, 他不需要看一行代码, 只要看文档字符串就可以了. 应该给参数单独写文档. 在冒号后跟上解释, 而且应该用统一的悬挂式2或4空格缩进. 文档字符串应该在需要特定类型的地方指定期望的类型. “Raise:”部分应该列出该函数可能触发的所有异常. 生成器函数的文档字符串应该用”Yields:”而非”Returns:”.

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

**类**

- 类应该在其定义下有一个用于描述该类的文档字符串．如果你的类有公共属性(Attributes)，那么文档中应该有一个属性(Attributes)段．并且应该遵守和函数参数相同的格式．

```
class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

**块注释和行注释**

- 最需要写注释的是代码中那些技巧性的部分．如果你在下次代码走查的时候必须解释一下，那么你应该现在就给它写注释．对于复杂的操作，应该在其操作开始前写上若干行注释．对于不是一目了然的代码，应在其行尾添加注释．

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:        # true iff i is a power of 2
```

为了提高可读性，注释应该至少离开代码 2 个空格．

另一方面，绝不要描述代码．假设阅读代码的人比你更懂 Python，他只是不知道你的代码要做什么．

# BAD COMMENT: Now go through the b array and make sure whenever i occurs
# the next element is i+1

# 类

- 如果一个类不继承自其它类，就显式的从 object 继承．嵌套类也一样.

No:
```
class SampleClass:
    pass


class OuterClass:

    class InnerClass:
        pass
```

Yes:
```
class SampleClass(object):
    pass


class OuterClass(object):

    class InnerClass(object):
        pass


class ChildClass(ParentClass):
    """Explicitly inherits from another class already."""
```

继承自 object 是为了使属性(properties)正常工作，并且这样可以保护你的代码，使其不受 Python 3000 的一个特殊的潜在不兼容性影响．这样做也定义了一些特殊的方法，这些方法实现了对象的默认语义，包括 __new__, __init__, __delattr__, __getattribute__, __setattr__, __hash__, __repr__, and __str__ .

# 字符串

- 用%操作符格式化字符串，即使参数都是字符串．不过也不能一概而论，你需要在+和%之间好好判定.

No:
```
x = '%s%s' % (a, b)  # use + in this case
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
```

Yes:
```
x = a + b
x = '%s, %s!' % (imperative, expletive)
x = 'name: %s; score: %d' % (name, n)
```

避免在循环中用+和+=操作符来累加字符串．由于字符串是不可变的，这样做会创建不必要的临时对象，并且导致二次方而不是线性的运行时间．作为替代方案，你可以将每个子串加入列表，然后在循环结束后用 .join 连接列表．(也可以将每个子串写入一个 cStringIO.StringIO 缓存中.)

No:
```
employee_table = '<table>'
for last_name, first_name in employee_list:
    employee_table += '<tr><td>%s, %s</td></tr>' % (last_name, first_name)
employee_table += '</table>'
```

Yes:
```
items = ['<table>']
for last_name, first_name in employee_list:
    items.append('<tr><td>%s, %s</td></tr>' % (last_name, first_name))
items.append('</table>')
employee_table = ''.join(items)
```

为多行字符串使用三重双引号而非三重单引号．不过要注意，通常用隐式行连接更清晰，因为多行字符串与程序其他部分的缩进方式不一致．

No:
```
    print """This is pretty ugly.
Don't do this.
"""
```

Yes:
```
  print ("This is much nicer.\n"
         "Do it this way.\n")
```

## TODO 注释

- 为临时代码使用 TODO 注释，它是一种短期解决方案．不算完美，但够好了．

TODO 注释应该在所有开头处包含”TODO”字符串，紧跟着是用括号括起来的你的名字，email 地址或其它标识符．然后是一个可选的冒号．接着必须有一行注释，解释要做什么．主要目的是为了有一个统一的 TODO 格式，这样添加注释的人就可以搜索到(并可以按需提供更多细节)．写了 TODO 注释并不保证写的人会亲自解决问题．

```
# TODO(kl@gmail.com): Drop the use of "has_key".
# TODO(Zeke) change this to use relations.
```

如果你的 TODO 是”将来做某事”的形式，那么请确保你包含了一个指定的日期(“2009 年 11 月解决”)或者一个特定的事件(“等到所有的客户都可以处理 XML 请求就移除这些代码”).

# 导入格式

- 每个导入应该独占一行

Yes: `import os`
`import sys`

No: `import os, sys`

导入总应该放在文件顶部，位于模块注释和文档字符串之后，模块全局变量和常量之前．导入应该按照从最通用到最不通用的顺序分组:

1. 标准库导入
2. 第三方库导入
3. 应用程序指定导入

每种分组中，应该根据每个模块的完整包路径按字典序排序，忽略大小写．

```
import foo
from foo import bar
from foo.bar import baz
from foo.bar import Quux
from Foob import ar
```

# 语句

- 通常每个语句应该独占一行

不过，如果测试结果与测试语句在一行放得下，你也可以将它们放在同一行．如果是 if 语句，只有在没有 else 时才能这样做．特别地，绝不要对 try/except 这样做，因为 try 和 except 不能放在同一行．

Yes:

```
if foo: bar(foo)
```

No:

```
if foo: bar(foo)
else:   baz(foo)

try:            bar(foo)
except ValueError: baz(foo)
```

```
try:
    bar(foo)
except ValueError: baz(foo)
```

## 访问控制

- 在 Python 中，对于琐碎又不太重要的访问函数，你应该直接使用公有变量来取代它们，这样可以避免额外的函数调用开销．当添加更多功能时，你可以用属性(property)来保持语法的一致性．

(译者注：重视封装的面向对象程序员看到这个可能会很反感，因为他们一直被教育：所有成员变量都必须是私有的！其实，那真的是有点麻烦啊．试着去接受 Pythonic 哲学吧)

另一方面，如果访问更复杂，或者变量的访问开销很显著，那么你应该使用像 get_foo() 和 set_foo() 这样的函数调用．如果之前的代码行为允许通过属性(property)访问 ，那么就不要将新的访问函数与属性绑定．这样，任何试图通过老方法访问变量的代码就没法运行，使用者也就会意识到复杂性发生了变化．

## 命名

```
module_name, package_name,
ClassName, method_name, ExceptionName, function_name,
GLOBAL_VAR_NAME,
instance_var_name, function_parameter_name, local_var_name.
```

**应该避免的名称**

1. 单字母名称，除了计数器和迭代器．
2. 包/模块名中的连字符(-)
3. 双下划线开头并结尾的名称(Python 保留，例如__init__)

**命名约定**

1. 所谓”内部(Internal)”表示仅模块内可用，或者，在类内是保护或私有的．
2. 用单下划线(_)开头表示模块变量或函数是 protected 的(使用 import * from 时不会包含)．
3. 用双下划线(__)开头的实例变量或方法表示类内私有．
4. 将相关的类和顶级函数放在同一个模块里．不像 Java，没必要限制一个类一个模块．
5. 对类名使用大写字母开头的单词(如 CapWords，即 Pascal 风格)，但是模块名应该用小写加下划线的方式(如 lower_with_under.py)．尽管已经有很多

现存的模块使用类似于 CapWords.py 这样的命名，但现在已经不鼓励这样做，因为如果模块名碰巧和类名一致，这会让人困扰.

Python 之父 Guido 推荐的规范

| Type | Public | Internal |
|---|---|---|
| Modules | lower_with_under | _lower_with_under |
| Packages | lower_with_under | |
| Classes | CapWords | _CapWords |
| Exceptions | CapWords | |
| Functions | lower_with_under() | _lower_with_under() |
| Global/Class Constants | CAPS_WITH_UNDER | _CAPS_WITH_UNDER |
| Global/Class Variables | lower_with_under | _lower_with_under |
| Instance Variables | lower_with_under | _lower_with_under (protected) or __lower_with_under (private) |
| Method Names | lower_with_under() | _lower_with_under() (protected) or __lower_with_under() (private) |
| Function/Method Parameters | lower_with_under | |
| Local Variables | lower_with_under | |

# Main

- 即使是一个打算被用作脚本的文件，也应该是可导入的. 并且简单的导入不应该导致这个脚本的主功能(main functionality)被执行，这是一种副作用. 主功能应该放在一个 main()函数中.

在 Python 中, pychecker, pydoc 以及单元测试要求模块必须是可导入的. 你的代码应该在执行主程序前总是检查 if __name__ == '__main__'，这样当模块被导入时主程序就不会被执行.

```
def main():
     ...

if __name__ == '__main__':
    main()
```

所有的顶级代码在模块导入时都会被执行．要小心不要去调用函数，创建对象，或者执行那些不应该在使用 pychecker 或 pydoc 时执行的操作.

# Python 语言规范

## pychecker

- **对你的代码运行 pychecker**

定义:

pychecker 是一个在 Python 源代码中查找 bug 的工具. 对于 C 和 C++这样的不那么动态的(译者注: 原文是 less dynamic)语言, 这些 bug 通常由编译器来捕获. pychecker 和 lint 类似. 由于 Python 的动态特性, 有些警告可能不对. 不过伪告警应该很少.

优点:

可以捕获容易忽视的错误, 例如输入错误, 使用未赋值的变量等.

缺点:

pychecker 不完美. 要利用其优势, 我们有时侯需要: a) 围绕着它来写代码 b) 抑制其告警 c) 改进它, 或者 d) 忽略它.

结论:

确保对你的代码运行 pychecker.

关于如何运行 pychecker 的更多信息, 参考 pychecker 主页

你可以设置一个叫做__pychecker__的模块级变量来抑制适当的告警. 例如:

__pychecker__ = 'no-callinit no-classattr'

采用这种抑制方式的好处是我们可以轻松查找抑制并回顾它们.

你可以使用 pychecker --help 来获取 pychecker 告警列表.

要抑制”参数未使用”告警, 你可以用”_”作为参数标识符, 或者在参数名前加”unused_”. 遇到不能改变参数名的情况, 你可以通过在函数开头”提到”它们来消除告警. 例如:

```
def foo(a, unused_b, unused_c, d=None, e=None):
    (d, e) = (d, e)    # 让 pychecker 不告警
    return a
```

理想情况下, 我们以后会扩展 pychecker 以确保你真的没有使用这些参数.

# 导入

- **仅对包和模块使用导入**

定义:

模块间共享代码的重用机制.

优点:

命名空间管理约定十分简单. 每个标识符的源都用一种一致的方式指示. x.Obj 表示 Obj 对象定义在模块 x 中.

缺点:

模块名仍可能冲突. 有些模块名太长, 不太方便.

结论:

使用 import x 来导入包和模块.

使用 form x import y , 其中 x 是包前缀, y 是不带前缀的模块名.

使用 form x import y as z, 如果两个要导入的模块都叫做 z 或者 y 太长了.

例如, 模块 sound.effects.echo 可以用如下方式导入:

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

导入时不要使用相对名称. 即使模块在同一个包中, 也要使用完整包名. 这能帮助你避免无意间导入一个包两次.

# 包

- **使用模块的全路径名来导入每个模块**

优点:

避免模块名冲突. 查找包更容易.

缺点:

部署代码变难, 因为你必须复制包层次.

结论:

所有的新代码都应该用完整包名来导入每个模块.

应该像下面这样导入:

```
# Reference in code with complete name.
import sound.effects.echo
```

```
# Reference in code with just module name (preferred).
from sound.effects import echo
```

# 异常

- **允许使用异常，但必须小心**

定义:

异常是一种跳出代码块的正常控制流来处理错误或者其它异常条件的方式.

优点:

正常操作代码的控制流不会和错误处理代码混在一起. 当某种条件发生时，它也允许控制流跳过多个框架. 例如，一步跳出 N 个嵌套的函数，而不必继续执行错误的代码.

缺点:

可能会导致让人困惑的控制流. 调用库时容易错过错误情况.

结论:

异常必须遵守特定条件:

1. 像这样触发异常: raise MyException("Error message") 或者 raise MyException . 不要使用两个参数的形式( raise MyException, "Error message" )或者过时的字符串异常( raise "Error message" ).
2. 模块或包应该定义自己的特定域的异常基类，这个基类应该从内建的 Exception 类继承. 模块的异常基类应该叫做”Error”.
   3. class Error(Exception):
   4. pass
5. 永远不要使用 except: 语句来捕获所有异常，也不要捕获 Exception 或者 StandardError，除非你打算重新触发该异常，或者你已经在当前线程的最外层(记得还是要打印一条错误消息). 在异常这方面, Python 非常宽容, except: 真的会捕获包括 Python 语法错误在内的任何错误. 使用 except: 很容易隐藏真正的 bug.
6. 尽量减少 try/except 块中的代码量. try 块的体积越大，期望之外的异常就越容易被触发. 这种情况下, try/except 块将隐藏真正的错误.
7. 使用 finally 子句来执行那些无论 try 块中有没有异常都应该被执行的代码. 这对于清理资源常常很有用，例如关闭文件.

# 全局变量

- **避免全局变量**

定义:

定义在模块级的变量.

优点:

偶尔有用.

缺点:

导入时可能改变模块行为，因为导入模块时会对模块级变量赋值.

结论:

避免使用全局变量，用类变量来代替．但也有一些例外:

1. 脚本的默认选项.
2. 模块级常量．例如:　PI = 3.14159．常量应该全大写，用下划线连接.
3. 有时候用全局变量来缓存值或者作为函数返回值很有用.
4. 如果需要，全局变量应该仅在模块内部可用，并通过模块级的公共函数来访问.

## 嵌套/本地/内部类或函数

- **鼓励使用嵌套/本地/内部类或函数**

定义:

类可以定义在方法，函数或者类中．函数可以定义在方法或函数中．封闭区间中定义的变量对嵌套函数是只读的.

优点:

允许定义仅用于有效范围的工具类和函数.

缺点:

嵌套类或局部类的实例不能序列化(pickled).

结论:

推荐使用.

## 列表推导(List Comprehensions)

- **可以在简单情况下使用**

定义:

列表推导(list comprehensions)与生成器表达式(generator expression)提供了一种简洁高效的方式来创建列表和迭代器，而不必借助 map(), filter()，或者 lambda.

优点:

简单的列表推导可以比其它的列表创建方法更加清晰简单．生成器表达式可以十分高效，因为它们避免了创建整个列表.

缺点:

复杂的列表推导或者生成器表达式可能难以阅读.

结论:

适用于简单情况．每个部分应该单独置于一行：映射表达式, for 语句，过滤器表达式.
禁止多重 for 语句或过滤器表达式．复杂情况下还是使用循环.

No:

```
  result = [(x, y) for x in range(10) for y in range(5) if x * y >
10]

  return ((x, y, z)
         for x in xrange(5)
         for y in xrange(5)
         if x != y
         for z in xrange(5)
         if y != z)
```

Yes:

```
  result = []
  for x in range(10):
      for y in range(5):
          if x * y > 10:
              result.append((x, y))

  for x in xrange(5):
      for y in xrange(5):
          if x != y:
              for z in xrange(5):
                  if y != z:
                      yield (x, y, z)

  return ((x, complicated_transform(x))
         for x in long_generator_function(parameter)
         if x is not None)

  squares = [x * x for x in range(10)]

  eat(jelly_bean for jelly_bean in jelly_beans
      if jelly_bean.color == 'black')
```

## 默认迭代器和操作符

- **如果类型支持，就使用默认迭代器和操作符．比如列表，字典及文件等．**

定义：

容器类型，像字典和列表，定义了默认的迭代器和关系测试操作符(in 和 not in)

优点：

默认操作符和迭代器简单高效，它们直接表达了操作，没有额外的方法调用．使用默认操作符的函数是通用的．它可以用于支持该操作的任何类型．

缺点：

你没法通过阅读方法名来区分对象的类型(例如, has_key()意味着字典). 不过这也是个有优点.

结论:

如果类型支持, 就使用默认迭代器和操作符, 例如列表, 字典和文件. 内建类型也定义了迭代器方法. 优先考虑这些方法, 而不是那些返回列表的方法. 除非你在遍历容器时不能修改它.

Yes:
```
for key in adict: ...
if key not in adict: ...
if obj in alist: ...
for line in afile: ...
for k, v in dict.iteritems(): ...
```

No:
```
for key in adict.keys(): ...
if not adict.has_key(key): ...
for line in afile.readlines(): ...
```

# 生成器

- **按需使用生成器.**

定义:

所谓生成器函数, 就是每当它执行一次生成(yield)语句, 它就返回一个迭代器, 这个迭代器生成一个值. 生成值后, 生成器函数的运行状态将被挂起, 直到下一次生成.

优点:

简化代码, 因为每次调用时, 局部变量和控制流的状态都会被保存. 比起一次创建一系列值的函数, 生成器使用的内存更少.

缺点:

没有.

结论:

鼓励使用. 注意在生成器函数的文档字符串中使用”Yields:”而不是”Returns:”.

# Lambda 函数

- **适用于单行函数**

定义:

与语句相反, lambda 在一个表达式中定义匿名函数. 常用于为 map()和 filter()之类的高阶函数定义回调函数或者操作符.

优点:

方便.

缺点:

比本地函数更难阅读和调试．没有函数名意味着堆栈跟踪更难理解．由于 lambda 函数通常只包含一个表达式，因此其表达能力有限．

结论:

适用于单行函数．如果代码超过 60-80 个字符，最好还是定义成常规(嵌套)函数．

# 默认参数值

- **适用于大部分情况．**

定义:

你可以在函数参数列表的最后指定变量的值，例如，def foo(a, b = 0): ．如果调用 foo 时只带一个参数，则 b 被设为 0．如果带两个参数，则 b 的值等于第二个参数．

优点:

你经常会碰到一些使用大量默认值的函数，但偶尔(比较少见)你想要覆盖这些默认值．默认参数值提供了一种简单的方法来完成这件事，你不需要为这些罕见的例外定义大量函数．同时，Python 也不支持重载方法和函数，默认参数是一种”仿造”重载行为的简单方式．

缺点:

默认参数只在模块加载时求值一次．如果参数是列表或字典之类的可变类型，这可能会导致问题．如果函数修改了对象(例如向列表追加项)，默认值就被修改了．

结论:

鼓励使用，不过有如下注意事项:

不要在函数或方法定义中使用可变对象作为默认值．

Yes:
```
def foo(a, b=None):
    if b is None:
        b = []
```

No:
```
def foo(a, b=[]):
    ...
```

调用方代码必须为带有默认值的参数使用带有名字的值．这多少能增加代码的可读性，并且当增加参数时能避免和检测接口被破坏．

```
def foo(a, b=1):
    ...
```

Yes:
```
foo(1)
foo(1, b=2)、
```

No:
```
foo(1, 2)
```

# 属性(properties)

- **访问和设置数据成员时，你通常会使用简单，轻量级的访问和设置函数．建议用属性来代替它们．**

定义:

一种用于包装方法调用的方式．当运算量不大，它是获取和设置属性(attribute)的标准方式．

优点:

通过消除简单的属性(attribute)访问时显式的 get 和 set 方法调用，可读性提高了．允许懒惰的计算．用 Pythonic 的方式来维护类的接口．就性能而言，当直接访问变量是合理的，添加访问方法就显得琐碎而无意义．使用属性(properties)可以绕过这个问题．将来也可以在不破坏接口的情况下将访问方法加上．

缺点:

属性(properties)是在 get 和 set 方法声明后指定，这需要使用者在接下来的代码中注意: set 和 get 是用于属性(properties)的(除了用@property 装饰器创建的只读属性)．必须继承自 object 类．可能隐藏比如操作符重载之类的副作用．继承时可能会让人困惑．

结论:

你通常习惯于使用访问或设置方法来访问或设置数据，它们简单而轻量．不过我们建议你在新的代码中使用属性．只读属性应该用 @property 装饰器来创建．

如果子类没有覆盖属性，那么属性的继承可能看上去不明显．因此使用者必须确保访问方法间接被调用，以保证子类中的重载方法被属性调用(使用模板方法设计模式)．

Yes:
```
import math

    class Square(object):
        """A square with two properties: a writable area and a
read-only perimeter.

        To use:
        >>> sq = Square(3)
        >>> sq.area
        9
        >>> sq.perimeter
        12
        >>> sq.area = 16
        >>> sq.side
        4
        >>> sq.perimeter
        16
        """
```

```
    def __init__(self, side):
        self.side = side

    def __get_area(self):
        """Calculates the 'area' property."""
        return self.side ** 2

    def ___get_area(self):
        """Indirect accessor for 'area' property."""
        return self.__get_area()

    def __set_area(self, area):
        """Sets the 'area' property."""
        self.side = math.sqrt(area)

    def ___set_area(self, area):
        """Indirect setter for 'area' property."""
        self._SetArea(area)

    area = property(___get_area, ___set_area,
                    doc="""Gets or sets the area of the square.""")

    @property
    def perimeter(self):
        return self.side * 4
```

(译者注：老实说，我觉得这段示例代码很不恰当，有必要这么蛋疼吗?)

# True/False 的求值

- **尽可能使用隐式 false**

定义:

Python 在布尔上下文中会将某些值求值为 false．按简单的直觉来讲，就是所有的”空”值都被认为是 false．因此 0， None, [], {}, “” 都被认为是 false.

优点:

使用 Python 布尔值的条件语句更易读也更不易犯错．大部分情况下，也更快.

缺点:

对 C/C++开发人员来说，可能看起来有点怪.

结论:

尽可能使用隐式的 false，例如：使用 if foo: 而不是 if foo != []:。不过还是有一些注意事项需要你铭记在心：

1. 永远不要用==或者!=来比较单件，比如 None。使用 is 或者 is not。
2. 注意：当你写下 if x: 时，你其实表示的是 if x is not None 。例如：当你要测试一个默认值是 None 的变量或参数是否被设为其它值。这个值在布尔语义下可能是 false!
3. 永远不要用==将一个布尔量与 false 相比较。使用 if not x: 代替。如果你需要区分 false 和 None，你应该用像 if not x and x is not None: 这样的语句。
4. 对于序列(字符串，列表，元组)，要注意空序列是 false。因此 if not seq: 或者 if seq: 比 if len(seq): 或 if not len(seq): 要更好。
5. 处理整数时，使用隐式 false 可能会得不偿失(即不小心将 None 当做 0 来处理)。你可以将一个已知是整型(且不是 len()的返回结果)的值与 0 比较。

Yes:
```
if not users:
    print 'no users'

if foo == 0:
    self.handle_zero()

if i % 10 == 0:
    self.handle_multiple_of_ten()
```

No:
```
if len(users) == 0:
    print 'no users'

if foo is not None and not foo:
    self.handle_zero()

if not i % 10:
    self.handle_multiple_of_ten()
```

6. 注意‘0’(字符串)会被当做 true。

## 过时的语言特性

- **尽可能使用字符串方法取代字符串模块。使用函数调用语法取代 apply()。使用列表推导, for 循环取代 filter(), map()以及 reduce()。**

定义：

当前版本的 Python 提供了大家通常更喜欢的替代品。

结论：

我们不使用不支持这些特性的 Python 版本，所以没理由不用新的方式。

No:  `words = string.split(foo, ':')`

`map(lambda x: x[1], filter(lambda x: x[2] == 5, my_list))`

`apply(fn, args, kwargs)`

Yes: `words = foo.split(':')`

`[x[1] for x in my_list if x[2] == 5]`

`fn(*args, **kwargs)`

# 静态 Scoping(Lexical Scoping)

- **推荐使用**

定义:

嵌套的 Python 函数可以引用外层函数中定义的变量，但是不能够对它们赋值．变量绑定的解析是使用静态 Scoping，也就是基于静态的程序文本．对一个块中的某个名称的任何赋值都会导致 Python 将对该名称的全部引用当做局部变量，甚至是赋值前的处理．如果碰到 global 声明，该名称就会被视作全局变量．

一个使用这个特性的例子:

```
def get_adder(summand1):
    """Returns a function that adds numbers to a given number."""
    def adder(summand2):
        return summand1 + summand2

    return adder
```

(译者注：这个例子有点诡异，你应该这样使用这个函数: sum = get_adder(summand1)(summand2) )

优点:

通常可以带来更加清晰，优雅的代码．尤其会让有经验的 Lisp 和 Scheme(还有 Haskell, ML 等)程序员感到欣慰．

缺点:

可能导致让人迷惑的 bug．例如下面这个例子:

```
i = 4
def foo(x):
```

```
    def bar():
        print i,
    # ...
    # A bunch of code here
    # ...
    for i in x:  # Ah, i *is* local to Foo, so this is what Bar sees
        print i,
    bar()
```

因此 foo([1, 2, 3]) 会打印 1 2 3 3，不是 1 2 3 4.

(译者注: x 是一个列表, for 循环其实是将 x 中的值依次赋给 i.这样对 i 的赋值就隐式的发生了，整个 foo 函数体中的 i 都会被当做局部变量，包括 bar()中的那个. 这一点与 C++之类的静态语言还是有很大差别的.)

结论:

鼓励使用.

## 函数与方法装饰器

- **如果好处很显然，就明智而谨慎的使用装饰器**

定义:

用于函数及方法的装饰器(也就是@标记). 最常见的装饰器是@classmethod 和 @staticmethod，用于将常规函数转换成类方法或静态方法. 不过，装饰器语法也允许用户自定义装饰器. 特别地，对于某个函数 my_decorator，下面的两段代码是等效的:

```
class C(object):
   @my_decorator
   def method(self):
      # method body ...
class C(object):
    def method(self):
        # method body ...
    method = my_decorator(method)
```

优点:

优雅的在函数上指定一些转换. 该转换可能减少一些重复代码，保持已有函数不变(enforce invariants)，等.

缺点:

装饰器可以在函数的参数或返回值上执行任何操作，这可能导致让人惊异的隐藏行为. 而且，装饰器在导入时执行. 从装饰器代码的失败中恢复更加不可能.

结论:

如果好处很显然，就明智而谨慎的使用装饰器．装饰器应该遵守和函数一样的导入和命名规则．装饰器的 python 文档应该清晰的说明该函数是一个装饰器．请为装饰器编写单元测试．

避免装饰器自身对外界的依赖(即不要依赖于文件，socket，数据库连接等)，因为装饰器运行时这些资源可能不可用(例如导入时，使用 pychecker 或其它工具时)．应该保证一个用有效参数调用的装饰器在所有情况下都是成功的．

装饰器是一种特殊形式的”顶级代码”．参考后面关于 Main 的话题．

# 线程

- **不要依赖内建类型的原子性．**

虽然 Python 的内建类型例如字典看上去拥有原子操作，但是在某些情形下它们仍然不是原子的(即：如果__hash__或__eq__被实现为 Python 方法)且它们的原子性是靠不住的．你也不能指望原子变量赋值(因为这个反过来依赖字典)．

优先使用 Queue 模块的 Queue 数据类型作为线程间的数据通信方式．另外，使用 threading 模块及其锁原语．了解条件变量的合适使用方式，这样你就可以使用 threading.Condition 来取代低级别的锁了．

# 威力过大的特性

- **避免使用这些特性**

定义:

Python 是一种异常灵活的语言，它为你提供了很多花哨的特性，诸如原类(metaclasses)，字节码访问，任意编译(on-the-fly compilation)，动态继承，对象父类重定义(object reparenting)，导入黑客(import hacks)，反射，系统内修改(modification of system internals)，等等．

优点:

强大的语言特性，能让你的代码更紧凑．

缺点:

使用这些很”酷”的特性十分诱人，但不是绝对必要．使用奇技淫巧的代码将更加难以阅读和调试．开始可能还好(对原作者而言)，但当你回顾代码，它们可能会比那些稍长一点但是很直接的代码更加难以理解．

结论:

在你的代码中避免这些特性．