From Elias的个人主页

Python: Python 编码风格指南中译版（Google SOC）

针对Python Style Guide Jun 18, 2009 版本翻译

- 译文发布于：http://www.elias.cn/Develop/PythonStyleGuide
- 译者：elias DOT soong AT gmail DOT com

本文是向 Melange 项目贡献 Python 代码的编码风格指南。

SoC framework项目以及在此之上构建的Melange web applications都是用 Python 语言实现的（因为 Python 是Google App Engine目前支持的唯一一种编程语言）。本编码风格指南是向 Melange 项目贡献 Python 代码的“要做”和“不要做”的列表。

下面这些规则不是原则或建议，而是严格的规定。你不能不理会这些规定除非是确实需要使用并被允许了，但仍然要注意本指南结尾处一致性部分的建议。

如果你有关于这些准则的问题，可以向开发者邮件列表发邮件询问。请在邮件标题写上“Python style:”，以便在邮件列表归档中更容易定位这些讨论。

本编码风格指南写在 Melange 项目的 Wiki 上，可以遵循已有的文档复查流程?进行修改。但不应该轻易修改本文，因为一致性是本指南的一个关键目标，而且不能因为新的风格指南变化而需要更新旧代码。

1. 概述

1.1 Python 语言方面的准则

1. pychecker: 建议使用
2. 导入模块和包: 可以，但不要 import *
3. 完整路径导入: 可以
4. 异常处理: 可以
5. 全局变量: 谨慎使用
6. 内嵌/本地/内部类和函数: 可以
7. List Comprehensions: 可以用，如果简明易懂的话
8. 默认迭代器和运算符: 可以
9. 生成器: 可以
10. 使用 apply、 filter、 map、 reduce: 对one-liner来说可以
11. Lambda 函数: 对one-liner来说可以

12. 默认参数值：可以
13. Properties：可以
14. True/False 求值：可以
15. 布尔内置类型：可以
16. String 方法：可以
17. 静态域：可以
18. 函数和方法修饰符：适度使用
19. 线程：不要用（Google App Engine不支持）
20. 高级特性：不要用

1.2 Python 编码风格方面的准则

所有文件都保持一致风格时程序要容易维护得多，以下是 Melange 项目的标准 Python 编码风格规范：

1. 分号：避免使用
2. 每行长度：最多80列
3. 圆括号：吝啬地用
4. 缩进：2 空格（不要用 tab ），和PEP8有所区别
5. 空行：对函数和类用 2 个空行分隔，对类的方法用 1 个空行
6. 空格：在行内吝啬地使用
7. Python 解释器：用Google App Engine支持的那个： #!/usr/bin/python2.5
8. 注释：__doc__ strings、块注释、行内注释
    - __doc__ Strings
    - 模块
    - 函数和方法
    - 类
    - 块注释及行内注释
9. 类：继承 object
10. 字符串：避免重复使用 + 和 +=
11. TODO style：TODO(username)：使用一种一致的风格
12. import 分组、排序和整理：一行一个，按包名字分组顺序放置，按字母顺序排序
13. 语句：一行一个，避免使用分号
14. 访问控制：轻量化风格可以用 foo，否则用 GetFoo() 和 SetFoo()
15. 命名：用 foo_bar.py 而不是 foo-bar.py，和PEP8有些区别
16. 程序入口：if __name__ == '__main__':
17. 总结：看看你周围是什么

2. Python 语言方面的准则

2.1 pychecker

是什么： pychecker 是从 Python 源代码中找 Bug 的工具。类似 lint ，它寻找在 C 和 C++ 那样更少动态化的语言中通常由编译器捕捉的那些问题。因为 Python 天生就是动态化的，其中一些警告可能并不正确；但假警报应该是相当少见的。

优点： 捕捉易犯的错误，比如拼写、在赋值之前就使用变量等等。

缺点： pychecker 不是完美的。为了好好利用它，我们有时候得：a）针对它来写代码 b）禁用特定警告 c）改进它 d）或者忽略它。

决定： 确保在代码上执行 pychecker。

你可以设置一个名为 __pychecker__ 的模块级别变量来适当禁用某些警告。比如：

```
__pychecker__ = 'no-callinit no-classattr'
```

[Get Code]

用这种方法禁用有个好处：我们可以很容易地找到这些禁用并且重新启用它们。

你可以用 pychecker --help 来获得 pychecker 警告的列表。

禁用“未被使用的变量”警告可以用 _ 作为未使用变量的标识符，或者给变量名添加 unused_ 前缀。有些情况下无法改变变量名，那你可以在函数开头调用它们一下，比如：

```
def Foo(a, unused_b, unused_c, d=None, e=None):
  d, e = (d, e)  # silence pychecker
  return a
```

[Get Code]

最理想的是， pychecker 会竭尽全力保证这样的“未使用声明”是真实的。

你可以在PyChecker 主页找到更多相关信息。

2.2 导入模块和包

是什么： 在模块与模块之间共享代码的重用机制。

优点： 最简单同时也是最常用的共用方法。

缺点： from foo import * 或者 from foo import Bar 很恶心而且这会使寻找模块之间的依赖关系变难，从而导致严重的维护问题。

决定： 用 import x 来导入包和模块。只有在 x 是一个包（package），而 y 是一个模块（module）的时候才用 from x import y 。这可以让使用者无需说明完整的包前缀就能引用模块。比如 sound.effects.echo 包可以像下面这样导入：

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

[Get Code]

即使模块是在同一个包里，也不要直接导入模块而不给出完整的包名字。这可能导致包被导入多次以及非预期的副作用。

例外的情况：当且仅当 Bar 是 foo 中的一个顶级 singleton，并且叫做 foo_Bar 是为了描述 Bar 与 foo 之间的关系，那么才可以用 from foo import Bar as foo_Bar 。

比如，像下面这样是可以的：

```
from soc.logic.models.user import logic as user_logic
...
user_logic.getForCurrentAccount()
```

[Get Code]

以下写法更受欢迎：

```
from soc.logic import models
```

```
...
import soc.logic.models.user
...
models.logic.user.logic.getForCurrentAccount()
```

[Get Code]

2.3 完整路径导入

是什么： 每个模块都通过模块的完整路径位置导入和引用。

优点： 避免模块名字之间的冲突，而且查找模块更容易。

缺点： 部署代码会麻烦一些，因为你必须重现整个包的分层结构。

决定： 所有新代码都应该使用他们的包名字来引用模块。不要为了从其他目录导入模块而修改 sys.path 和 PATHONPATH。（可能有些情况下仍然需要修改路径，但只要可能的话就应该避免这样做。）

应该像下面这样导入：

```
# 使用完整名字引用：
import soc.logging

# 仅使用模块名字引用：
from soc import logging
```

[Get Code]

有些包名字可能和常用的本地变量名是一样的。在这种情况下为了避免混乱，先导入包然后清晰地单独导入模块有时候也是有意义的。结果看起来像这样：

```
from soc import models
import soc.models.user

...
  user = models.user.User()
```

[Get Code]

在创建时就避免易引发混乱的模块命名可能是最佳方案，但有时选择直观的模块名字（比如上面例子中 soc.models 中的那些模块）会导致需要像上面这样做 workaround 。

2.4 异常处理

是什么： 异常处理指的是为处理错误及其他异常状况而打破代码块的正常控制流。

优点： 正常操作代码的控制流不会被错误处理代码弄乱。也可以在特定情况发生时让控制流跳过多个步骤，比如一步就从 N 层嵌套函数（nested functions）中返回，而不必继续把错误代码一步步执行到底。

缺点： 可能使控制流变得难以理解，以及在做函数库调用时容易忽略一些错误情况。

决定： 异常处理是很 Pythonic 的，因此我们当然同意用它，但只是在符合以下特定条件时：

- 要像这样抛出异常：raise MyException("Error message") 或者 raise MyException，而不要用双参数的形式（raise MyException, "Error message"） 或者 已废弃的基于字符串的异常（raise "Error message"）。
- 模块和包应该定义自己的特定领域的基础异常类，而且这个类应该继承自内置的 Exception 类。这种用于一个模块的基础异常应该命名为 Error。

```
class Error(Exception):
  """Base exception for all exceptions raised in module Foo."""
  pass
```

[Get Code]

- 永远不要直接捕获所有异常，也即 except: 语句，或者捕获 Exception 和 StandardError，除非你会把异常重新抛出或者是在你线程最外层的代码块中（而且你会打印一个出错信息）。在这种意义上来讲， Python 是很健壮的，因为 except: 会真正捕获包括 Python 语法错误在内的所有东西。使用 except: 很容易会掩盖真正的 Bug 。
- 在 try/except 块中使代码量最小化。try 里头的内容越多，就更有可能出现你原先并未预期会抛出异常的代码抛出异常的情况。在这种情况下，try/except 代码块就隐藏了真正的错误。
- 使用 finally 子句执行一段代码而无论 try 语句块是否会抛出异常。这在做清除工作的时候经常是很有用的，比如关闭文件。

2.5 全局变量

是什么： 在模块级别声明的变量。

优点： 有时很有用。

缺点： 有可能在导入时改变模块的行为，因为在模块被导入时完成了模块级别变量的赋值。

决定： 推荐使用类变量而避免使用全局变量。以下是一些例外情况：

- 作为脚本中的默认选项。
- 模块级别常量，例如 PI = 3.14159 。常量名应该全部使用大写字母并用下划线分隔，参考下文命名章节。
- 有时全局变量对缓存函数返回值或其他需要的运算结果会是挺有用的。
- 在需要的时候，全局变量应该被用作模块的内部变量，并通过公开的模块级函数来访问。参考下文命名章节。

2.6 内嵌/本地/内部类和函数

是什么： 类可以定义在函数或另一个类的内部。函数可以定义在另一个函数的内部。内嵌函数可以只读访问定义在外部作用域中的变量。

优点： 允许定义只用于一个非常有限的作用域内部的工具类和工具函数，实在太方便了（ADT-y）。

缺点： 内嵌的本地类实例无法被序列化导出（pickle）。

决定： 挺好的，用吧。

2.7 List Comprehensions

是什么： List Comprehension 和 Generator Expression 提供了一种简明高效的方法来创建列表和迭代器而无需借助 map() 、 filter() 、或者 lambda 。

优点： 简单的 List Comprehensions 比其他列表创建技术更清晰、更简单。Generator Expressions 非常高效，因为它避免了一下子创建整个列表。

缺点： 复杂的 List Comprehensions 或 Generator Expressions 很难读懂。

决定： 对一行风格（one-liner）来说没问题，或者在 `x for y in z` 能写在一行里而条件语句可以写成另一行的时候（注意下面 `x` 很长的、写成三行的那个例子是可以接受的）也可以用。当问题变得更复杂时应该用循环来代替。这是一种主观判断。

No:

```
# 太复杂了，应该用循环嵌套代替：
result = [(x, y) for x in range(10)
                 for y in range(5)
                   if x * y > 10]
```

[Get Code]

Yes:

```
# 这个例子足够复杂了，*不应该*使用List Comprehension：
result = []
for x in range(10):
  for y in range(5):
    if x * y > 10:
      result.append((x, y))

# 简单舒服的一行风格：
squares = [x * x for x in range(10)]

# 写成两行的 Generator （不是 List Comprehension）例子，也是可以的：
Eat(jelly_bean for jelly_bean in jelly_beans
    if jelly_bean.color == 'black')

# 写成三行的这个例子是否仍然比等价的循环具有更好的可读性还存在争议：
result = [someReallyLongWayToEatAJellyBean(jelly_bean)
          for jelly_bean in jelly_beans
          if jelly_bean != 'black']
```

[Get Code]

2.8 默认迭代器和运算符

是什么： 容器类型（比如字典和列表）定义了默认迭代器和从属关系测试运算符（例如 `in` 和 `not in`）。

优点： 默认迭代器和运算符又简单又高效，它们无需额外方法调用就能直接表达操作。使用默认运算符来编写函数是一种通用性较强的做法，因为它可以用于任何支持这些操作的数据类型。

缺点： 你无法通过阅读方法名字就说出对象的类型（比如说 `has_key()` 表示一个字典）。这同时也是一个优点。

决定： 在支持它们的类型上使用默认迭代器和运算符，比如列表、字典、和文件。当然这些内置类型同时也定义了迭代器方法。对返回列表的方法倾向于使用迭代器方法，但你不应该在迭代的过程中修改容器中的内容。

Yes:

```
for key in adict: ...

if key not in adict: ...

if obj in alist: ...

for line in afile: ...

for k, v in dict.iteritems(): ...
```

[Get Code]

No:

```
for key in adict.keys(): ...
if not adict.has_key(key): ...
for line in afile.readlines(): ...
```

[Get Code]

2.9 生成器

是什么： 生成器函数返回一个迭代器，而这个迭代器每次执行 yield 语句生成一个计算结果。每次生成一个计算结果之后，生成器函数的运行时状态被挂起，直到需要生成下一个计算结果时。

优点： 代码更简单。因为对每次调用来说，本地变量和控制流的状态都是被保留的。生成器比用计算结果一次性填满整个列表的函数更节省内存。

缺点： 没有。

决定： 很好。在生成器函数的 __doc__ String 中使用 "Yields:" 而不是 "Returns:" 。

2.10 使用 apply filter map reduce

是什么： 实用的内置列表操作函数。通常和 lambda 函数一起用。

优点： 代码很紧凑。

缺点： 高阶函数式编程恐怕更难理解。

决定： 对简单代码和喜欢把代码写成一行的人来说，尽可能用 List Comprehension 而限制使用这些内置函数。一般来说，如果代码在任何地方长度超过60到80个字符，或者使用了多层函数调用（例如，map(lambda x: x[1], filter(...))），那这就是一个信号提醒你最好把它重新写成一个普通的循环。比较：

map/filter:

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

[Get Code]

list comprehensions:

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

[Get Code]

2.11 Lambda functions

是什么： Lambda 定义仅包含表达式、没有其他语句的匿名函数，通常用于定义回调或者用于 map() 和 filter() 这样高阶函数的运算符。

优点： 方便。

缺点： 比本地函数更难阅读和调试。没有名字意味着堆栈追踪信息更难理解。而且函数只能包含一个表达式，因而表达能力也比较有限。

决定： 对喜欢把代码写成一行的人来说没什么问题。只要 lambda 函数中的代码长度超过60到80个字符，那可能把它定义成一个标准（或者嵌套的）函数更合适。

对加法这样的普通操作，应该使用 operator 模块中的函数而不是用 lambda 函数。例如，应该用 operator.add 而不是 lambda x, y: x + y。（内置的 sum() 函数其实比这两者中的任何一个都更合适。）

2.12 默认参数值

是什么： 你可以给函数的参数列表中最靠后的几个变量指定取值，比如 def foo(a, b=0): 。如果只用一个参数来调用 foo ，b 将被设置为 0 。如果用两个参数来调用它， b 将使用第二个参数的值。

优点： 通常你会大量使用函数的默认值，但偶尔会需要覆盖这些值。默认参数值允许用一种简单的办法来做到这一点，而不必为偶尔的例外情形定义一大堆函数。而且， Python 不支持方法/函数重载，而参数默认值是“模仿”重载行为的一种简单方法。

缺点： 默认参数值会在模块加载的时候被赋值。如果参数是一个列表或者字典之类的可变对象就有可能造成问题。如果函数修改了这个对象（比如在列表最后附加了一个新的项），那默认值也就改变了。

决定： 在以下限制条款下是可以使用的：

- 不要把可变对象当作函数或方法定义的默认值。

Yes:

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

[Get Code]

No:

```
def foo(a, b=[]):
  ...
```

[Get Code]

调用函数的代码必须对默认参数使用指名赋值（named value）。这多少可以帮助代码的文档化，并且当增加新参数进来时帮助避免和发现破坏原有接口。

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

[Get Code]

Yes:

```
foo(1)
foo(1, b=2)
```

[Get Code]

No:

```
foo(1, 2)
```

[Get Code]

2.13 Properties

是什么： 在运算量很小时，把对属性的 get 和 set 方法调用封装为标准的属性访问方式的一个方法。

优点： 对简单的属性访问来说，去掉直率的 get 和 set 方法调用提高了代码的可读性。允许延后计算。考虑到 Pythonic 的方法来维护类的接口。在性能方面，当直接变量访问是合理的，允许 Properties 就省略了琐碎的属性访问方法，而且将来仍然可以在不破坏接口的情况下重新加入属性访问方法。

缺点： Properties 在 getter 和 setter 方法声明之后生效，也就是要求使用者注意这些方法在代码很靠后的地方才能被使用（除了用 @property 描述符创建的只读属性之外 —— 见下文详述）。必须继承自 object 。会像运算符重载一样隐藏副作用。对子类来说会很难理解。

决定： 在那些你通常本来会用简单、轻量的访问/设置方法的代码中使用 Properties 来访问和设置数据。只读的属性应该用 @property 描述符?来创建。

如果 Property 自身没有被覆盖，那 Properties 的继承并非显而易见。因此使用者必须确保访问方法被间接调用，以便确保子类中被覆盖了的方法会被 Property 调用（使用“模板方法（Template Method）”设计模式）。

Yes:

```
import math

class Square(object):
  """基类 square 有可写的 'area' 属性和只读的 'perimeter' 属性。

  可以这样使用：
  >>> 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 _getArea(self):
    """计算 'area' 属性的值"""
    return self.side ** 2

  def __getArea(self):
    """对 'area' 属性的间接访问器"""
    return self._getArea()

  def _setArea(self, area):
    """对 'area' 属性的设置器"""
    self.side = math.sqrt(area)
```

```
  def __setArea(self, area):
    """对 'area' 属性的间接设置器"""
    self._setArea(area)

  area = property(__getArea, __setArea,
                  doc="""Get or set the area of the square""")

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

[Get Code]

== True/False 求值

是什么： 在布尔型上下文下， Python 把一些特定的取值当作“false”处理。快速的“经验法则”是所有的“空”值都会被认为是“false”，也即 0、None、[]、{}、"" 在布尔型上下文中都会被当作“false”。

优点： 使用 Python 的布尔型条件更容易阅读而且更不容易产生错误。在大多数情况下，这也是运行速度更快的选择。

缺点： 对 C/C++ 开发者来说可能会看起来很奇怪。

决定： 在所有可能的情况下使用这种“隐含”的 false ，比如用 `if foo:` 而不是 `if foo != []:` 。这有几条你应该时刻注意的限制条件：

- 与具有唯一性的值比如 None 进行比较时总是应该使用 is 或者 is not。而且，留神在写 `if x:` 而你的实际意思是 `if x is not None:` 的时候，比如，测试一个默认值是 None 的变量或参数是否被设置为其他值。这里的“其他值”就有可能是在布尔型上下文中被认为是 false 的值！
- 对序列（strings、 lists、 tuples）来说，可以利用空序列就是 false 这一事实，也就是说 `if not seq:` 或者 `if seq:` 比 `if len(seq):` 或者 `if not len(seq):` 这种形式要更好。
- 注意 '0' （也即 0 当作字符串）会被求值为 true。

2.14 布尔内置类型

是什么： 从 Python 2.3 开始加入了布尔类型，也即加入了两个新的内置常量：True 和 False。

优点： 这使代码更容易阅读而且与之前版本中使用整数作为布尔型的做法向后兼容。

缺点： 没有。

决定： 使用布尔型。

2.15 String 方法

是什么： String 对象包括一些以前是 string 模块中的函数的方法。

优点： 无需导入 string 模块，而且这些方法在标准 byte 字符串和 unicode 字符串上都能使用。

缺点： 没有。

决定： 用吧。 string 模块已经被废弃了，现在更推荐使用 String 方法。

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

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

2.16 静态域

是什么： 被嵌套的 Python 函数（nested Python function）可以引用定义在容器函数（enclosing function）中的变量，但无法对它们重新赋值。变量绑定依据静态域（Lexical Scoping）决定，也就是说，基于静态的程序文本（static program text）。代码块中对一个名字的任意赋值都将导致 Python 把对这个名字的所有引用当作本地变量，即使是先调用后赋值。如果存在全局变量声明，那这个名字就会被当作是全局变量。

以下是使用这一特性的例子：

```
def getAdder(summand1):
  """getAdder 返回把数字与一个给定数字相加的函数。"""
  def anAdder(summand2):
    return summand1 + summand2

  return anAdder
```

[Get Code]

优点： 一般会得到更清晰、更优雅的代码。而且特别符合有经验的 Lisp 和 Scheme （以及 Haskell、ML 等等）程序员的习惯。

缺点： 没有。

决定： 可以用。

2.17 函数和方法修饰符

是什么： 在 Python 2.4 版本增加了函数和方法的修饰符（又称“@ notation”）。最常用的修饰符是 @classmethod 和 @staticmethod，用于把普通的方法转化为类方法或静态方法。然而，修饰符语法同样允许用户自定义修饰符。具体地，对函数 myDecorator 来说：

```
class C(object):
  @myDecorator
  def aMethod(self):
    # method body ...
```

[Get Code]

和如下代码是等价的：

```
class C(object):
  def aMethod(self):
    # method body ...
  aMethod = myDecorator(aMethod)
```

[Get Code]

优点： 能够优雅地对指定方法进行变形，而且这种变形避免了重复代码，强化了通用性（enforce invariants）等。

缺点： 修饰符能对函数的参数和返回值进行任意操作，会导致出乎意料之外的隐含行为。除此之外，修饰符会在导入阶段被执行。修饰符代码中的故障几乎是不可能被修复的。

决定： 在有明显好处时使用修饰符是明智的。修饰符应该和函数遵循同样的导入和命名准则。修饰符的 __doc__ string 应该清晰地声明该函数是一个修饰符，而且要为修饰符写单元测试。

避免在修饰符资深引入外部依赖关系（比如，不要依赖文件、 sockets 、数据库连接等），因为在修饰符运行时（在导入阶段，也许源于 pychecker 或其他工具）这些都有可能是无效的。在所有情况下都应（尽可能）保证使用有效参数调用修饰符时能成功运行。

修饰符是“顶级代码”（top level code）的一种特例 —— 参考主入口章节的更多讨论。

2.18 线程

Google App Engine 不支持线程，所以在 SoC framework 和 Melange Web 应用程序中也别用它。

2.19 高级特性

是什么： Python 是一种极为灵活的语言，提供诸如 metaclass、 bytecode 访问、即时编译、动态继承、 object reparenting、 import hacks、反射、修改系统内部结构等很多很炫的特性。

优点： 这些都是很强大的语言特性，能使你的代码更紧凑。

缺点： 在并非绝对必要的时候使用这些很“酷”的特性是很诱人的。里面使用了这些并不常见的特性的代码会更难读、更难懂、更难 debug。也许在刚开始的时候好像还没这么糟（对代码的原作者来说），但当你重新回到这些代码，就会觉得这比长点但简单直接的代码要更难搞。

决定： 在 Melange 代码中避免使用这些特性。例外的情形在开发者邮件列表中讨论。

3. Python 编码风格方面的准则

3.1 分号

不要用分号作为你的行结束符，也不要利用分号在同一行放两个指令。

3.2 每行长度

一行最多可以有80个字符。

例外： 导入模块的行可以超过80个字符再结束。

确保 Python 隐含的连接行（line joining）放在圆括号、方括号或大括号之间。如果需要的话，你可以在表达式两头放一堆额外的圆括号。

Yes:

```
fooBar(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')):
```

[Get Code]

当表示文本的字符串（literal string）一行放不下的时候，用圆括号把隐含的连接行括起来。

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

[Get Code]

注意上例中连续行中元素的缩进，参考缩进章节的解释。

3.3 圆括号

吝啬地使用圆括号。在以下情况下别用：
- 在 return 语句中。
- 在条件判断语句中。除非是用圆括号来暗示两行是连在一起的（参见上一节）。
- 在元组（tuple）周围。除非因为语法而不得不加或是为了显得更清晰。

但在下列情况下是可以用圆括号的：
- 暗示两行是连在一起的。
- 用来括起长表达式中的子表达式（包括子表达式是条件判断语句一部分的情形）。

实际上，在子表达式周围用圆括号比单纯依赖运算符优先级要好。

Yes：

```
if foo:
while x:
if x and y:
if not x:
if (x < 3) and (not y):
return foo
for x, y in dict.items():
x, (y, z) = funcThatReturnsNestedTuples()
```

[Get Code]

No：

```
if (x):
while (x):
if not(x):
if ((x < 3) and (not y)):
return (foo)
for (x, y) in dict.items():
(x, (y, z)) = funcThatReturnsNestedTuples()
```

[Get Code]

3.4 缩进

注意和PEP8不同，这里遵循作为本文起源的原始 Google Python 编码风格指南。

用两空格缩进代码块。不要用 tab 或者混用 tab 和空格。如果要暗示两行相连，要么就把被包装的元素纵向对其，就像“每行长度”章节中的例子那样；要么就用4个空格（不是两个，这样可以和后面紧跟着的嵌套代码块区分开，避免混淆）作悬挂缩进（hanging indent），在这种情况下，首行不应该放任何参数。

Yes：

```
# 与起始定界符对齐：
foo = longFunctionName(var_one, var_two,
                       var_three, var_four)

# 4空格悬挂缩进，而且首行空着：
foo = longFunctionName(
    var_one, var_two, var_three,
    var_four)
```

[Get Code]

No：

```
# 不应该在首行塞东西：
foo = longFunctionName(var_one, var_two,
    var_three, var_four)

# 不应该用两空格的悬挂缩进：
foo = longFunctionName(
  var_one, var_two, var_three,
  var_four)
```

[Get Code]

3.5 空行

在顶级定义（可以是函数或者类定义）之间加两个空行。在方法定义之间以及“ class ”那一行与第一个方法之间加一个空行。在__doc__ string和它后面的代码之间加一个空行。在函数和方法内部你认为合适的地方加一个空行。

在文件最后总是加一个空行，这可以避免很多 diff 工具生成“No newline at end of file”信息。

3.6 空格

在圆括号、方括号、大括号里面不要加空格。

Yes： spam(ham[1], {eggs: 2}, [])

No： spam(ham[1], { eggs: 2 }, [ ])

在逗号、分号、冒号前面不要加空格。逗号、分号、冒号后面必须加空格，除非那是行尾。

Yes：

```
if x == 4:

print x, y
```

```
x, y = y, x
```

[Get Code]

No:

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

[Get Code]

在表示参数、列表、下标、分块开始的圆括号/方括号前面不要加空格。

Yes: spam(1)

No: spam (1)

Yes: dict['key'] = list[index]

No: dict ['key'] = list [index]

在二元运算符两边各家一个空格，包括：赋值（=）、比较（==、<、>、!=、<>、<=、>=、in、not in、is、is not）、以及布尔运算符（and、or、not）。你肯定能判断出是否应该在算术运算符周围加空格，因为在二元运算符两边加空格的原则总是一致的。

Yes: x == 1

No: x<1

等号（“=”）用于指名参数或默认参数值时，两边不要加空格。

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)

3.7 Python 解释器

模块开头应该是一个“shebang”行，用于指定执行此程序的 Python 解释器：

```
#!/usr/bin/python2.5
```

[Get Code]

Google App Engine 要求使用 Python 2.5。

3.8 注释

Doc strings

Python 有一种独特的注释风格称为 __doc__ String。包、模块、类、函数的第一个语句如果是字符串那么就是一个 __doc__ String。这种字符串可以用对象的 __doc__() 成员函数自动提取，而且会被用于 pydoc。（试着在你的模块上运行 pydoc 来看看它是怎么工作的。）我们对 __doc__ String 的约定是：用三个双引号把字符串括起来。 __doc__ String 应该这样组织：首先是一个

以句号结束的摘要行（实实在在的一行，不多于80个字符），然后接一个空行，再接 __doc__ String 中的其他内容，并且这些文字的起始位置要和首行的第一个双引号在同一列。下面几节是关于 __doc__ String 格式更进一步的原则

模块

每个文件开头都应该包含一个带有版权信息和许可声明的块注释。

版权和许可声明

```
#!/usr/bin/python2.5
#
# Copyright [current year] the Melange authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#   http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

[Get Code]

然后接着写描述模块内容的__doc__ String，其中作者信息应该紧跟着许可声明。如果还给出了作者邮件地址，那添加到 __authors__ 列表的整个作者字符串应该就是一个与 RFC 2821 兼容的电子邮件地址。

模块头及作者信息

```
"""用一行文字概述模块或脚本，用句号结尾。

留一个空行。本 __doc__ string 的其他部分应该包括模块或脚本的全面描述。作为可选项，还可以包括导出的类和函数的简要描述。

  ClassFoo: 一行概述。
  functionBar(): 一行概述。
"""

__authors__ = [
  # 请按照姓氏字母顺序排列：
  '"John Smith" <johnsmith@example.com>',
  '"Joe Paranoid" <joeisgone@example.com>',  # 应提供电子邮件地址
]
```

[Get Code]

如果新的贡献者还没添加到AUTHORS file中，那应该在该贡献者第一次向版本控制工具提交代码时同时把他/她添加到这个文件里。

函数和方法

如果不是用途非常明显而且非常短的话，所有函数和方法都应该有 __doc__ string 。此外，所有外部能访问的函数和方法，无论有多短、有多简单，都应该有 __doc__ string 。 __doc__ string 应该包括函数能做什么、输入数据的具体描述（“Args:”）、输出数据的具体描述（“Returns:”、“Raises:”、或者“Yields:”）。 __doc__ string 应该能提供调用此函数相关的足够信息，而无需让使用者看函数的实现代码。如果参数要求特定的数据类型或者设置了参数默认值，那 __doc__ string 应该明确说明这两点。“Raises:”部分应该列出此函数可能抛出的所有异常。生成器函数的 __doc__ string 应该用“Yields:”而不是Returns:。

函数和方法的 __doc__ string 一般不应该描述实现细节，除非其中涉及非常复杂的算法。在难以理解的代码中使用块注释或行内注释会是更合适的做法。

```
def fetchRows(table, keys):
  """取出表中的多行内容。

  Args:
    table: 打开的表。 Table 类的实例。
    keys: 字符串序列，表示要从表中取出的行的键值。

  Returns:
    一个字典，映射指定键值与取出的表中对应行的数据：

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

    如果 keys 参数中的键值没有出现在字典里，就表示对应行在表中没找到。

  Raises:
    IOError: 访问 table.Table 对象时发生的错误。
  """
  pass
```

[Get Code]

类

类应该在描述它的类定义下面放 __doc__ string 。如果你的类有公开属性值，那应该在 __doc__ string 的 Attributes: 部分写清楚。

```
class SampleClass(object):

  """这里是类的概述。

  详细的描述信息……
  详细的描述信息……

  Attributes:
    likes_spam: 布尔型，表示我们是否喜欢垃圾邮件。
    eggs: 整数，数我们下了多少蛋。
  """

  def __init__(self, likes_spam=False):
    """拿点什么来初始化 SampleClass 。

    Args:
      likes_spam: 初始化指标，表示 SampleClass 实例是否喜欢垃圾邮件（默认是 False）。
    """
    self.likes_spam = likes_spam
    self.eggs = 0

  def publicMethod(self):
    """执行一些操作。"""
    pass
```

[Get Code]

块注释及行内注释

加注释的最后一个位置是在难以理解的代码里面。如果你打算在下一次代码复查（code review）的时候解释这是什么意思，那你应该现在就把它写成注释。在开始进行操作之前，就应该给复杂的操作写几行注释。对不直观的代码则应该在行末写注释。

```
# 我们用带权的字典检索来查找 i 在数组中的位置。我们根据数组中最大的数和
# 数组的长度来推断可能的位置，然后做二分法检索找到准确的值。

if i & (i-1) == 0:        # 当且仅当 i 是 2 的幂时，值为 true
```

[Get Code]

这些注释应该和代码分开才更易读。在块注释值钱应该加一个空行。一般行末注释应该和代码之间至少空两个格。如果连续几行都有行末注释（或者是在一个函数中），可以把它们纵向对齐，但这不是必须的。

另一方面，不要重复描述代码做的事。假设正在读代码的人比你更懂 Python （尽管这不是你努力的方向）。 On the other hand, never describe the code. Assume the person reading the code knows Python (though not what you're trying to do) better than you do.

```
# *不好的注释*：现在要遍历数组 b 并且确保任何时候 i 出现时，下一个元素都是 i+1
```

[Get Code]

3.9 类

如果不从其他基类继承，那就应该明确地从 object 基类继承。这一条对嵌套类也适用。

No:

```
class SampleClass:
  pass

class OuterClass:
  class InnerClass:
    pass
```

[Get Code]

Yes:

```
class SampleClass(object):
  pass

class OuterClass(object):
  class InnerClass(object):
    pass

class ChildClass(ParentClass):
  """已经从另一个类显式继承了。"""
  pass
```

[Get Code]

从 object 继承是为了让类属性能够正常工作，这会避免我们一旦切换到 Python 3000 时，打破已经习惯了的特有风格。同时这也定义了一些特殊函数，来实现对象（object）的默认语义，包括：__new__、__init__、__delattr__、__getattribute__、__setattr__、__hash__、__repr__、和 __str__。

3.10 字符串

应该用 % 运算符来格式化字符串，即使所有的参数都是字符串。不过你也可以在 + 和 % 之间做出你自己最明智的判断。

No:

```
x = '%s%s' % (a, b)  # 这种情况下应该用 +
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
```

[Get Code]

Yes:

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

[Get Code]

应该避免在循环中用 + 或 += 来连续拼接字符串。因为字符串是不变型，这会毫无必要地建立很多临时对象，从而二次方级别的运算量而不是线性运算时间。相反，应该把每个子串放到 list 里面，然后在循环结束的时候用 ''.join() 拼接这个列表。

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>'
```

[Get Code]

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)
```

[Get Code]

对多行字符串使用 """ 而不是 '''。但是注意，通常使用隐含的行连接（implicit line joining）会更清晰，因为多行字符串不符合程序其他部分的缩进风格。

No:

```
print """这种风格非常恶心。
不要这么用。
"""
```

[Get Code]

Yes:

```
print ("这样会好得多。\n"
       "所以应该这样用。\n")
```

[Get Code]

3.11 TODO style

在代码中使用 TODO 注释是临时的、短期的解决方案，或者说是足够好但不够完美的办法。

TODO 应该包括全部大写的字符串 TODO ，紧接用圆括号括起来的你的用户名： TODO(username) 。其中冒号是可选的。主要目的是希望有一种一致的 TODO 格式，而且可以通过用户名检索。

```
# TODO(someuser): 这里应该用 "*" 来做级联操作。
# TODO(anotheruser) 用 relations 来修改这儿。
```

[Get Code]

如果你的 TODO 是“在未来某个时间做某事”的形式，确保你要么包括非常确定的日期（“Fix by November 2008”）或者非常特殊的事件（“在数据库中的 Foo 实体都加上新的 fubar 属性之后删除这段代码。”）。

3.12 import 分组及顺序

应该在不同的行中做 import：

Yes:

```
import sys
import os
```

[Get Code]

No:

```
import sys, os
```

[Get Code]

import 总是放在文件开头的，也即在所有模块注释和 __doc__ string 的后面，在模块全局变量及常量的前面。import 应该按照从最常用到最不常用的顺序分组放置：

- import 标准库
- import 第三方库
- import Google App Engine 相关库
- import Django 框架相关库
- import SoC framework 相关库
- import 基于 SoC 框架的模块
- import 应用程序特有的内容

应该按照字母顺序排序，但所有以 from ... 开头的行都应该靠前，然后是一个空行，再然后是所有以 import ... 开头的行。以 import ... 开头的标准库和第三方库的 import 应该放在最前面，而且和其他分组隔开：

```
import a_standard
import b_standard
import a_third_party
import b_third_party

from a_soc import f
from a_soc import g

import a_soc
import b_soc
```

[Get Code]

在 import/from 行中，语句应该按照字母顺序排序：

```
from a import f
from a import g
from a.b import h
from a.d import e

import a.b
import a.b.c
import a.d.e
```

[Get Code]

3.13 语句

一般一行只放一个语句。但你可以把测试和测试结果放在一行里面，只要这样做合适。具体来说，你不能这样写 try/except ，因为 try 和 except 不适合这样，你只可以对不带 else 的 if 这么干。

Yes:

```
if foo: fuBar(foo)
```

[Get Code]

No:

```
if foo: fuBar(foo)
else:   fuBaz(foo)

try:                fuBar(foo)
except ValueError: fuBaz(foo)

try:
  fuBar(foo)
except ValueError: fuBaz(foo)
```

[Get Code]

3.14 访问控制

如果存取器函数很简单，那你应该用公开的变量来代替它，以避免 Python 中函数调用的额外消耗。在更多功能被加进来时，你可以用 Property 特性来保持语法一致性。

另一方面，如果访问更复杂，或者访问变量的成本较为显著，那你应该用函数调用（遵循命名准则），比如 getFoo() 或者 setFoo()。如果过去的行为允许通过 Property 访问，那就别把新的存取器函数绑定到 Property 上。所有仍然企图用旧方法访问变量的代码应该是非常显眼的，这样调用者会被提醒这种改变是比较复杂的。

3.15 命名

要避免的命名方式

- 使用单个字符命名，除非是计数器或迭代器。
- 在任何包或模块的名字里面使用减号。
- __double_leading_and_trailing_underscore__ 在变量开头和结尾都使用两个下划线（在 Python 内部有特殊含义）。

命名约定

注意这里某些命名约定和PEP8不一样，而是遵循“Google Python 编码风格指南”的原始版本，也即本编码风格指南的起源。

- “Internal”表示模块内部或类中的保护域（protected）和私有域。
- 变量名开头加一个下划线（_）能对保护模块中的变量及函数提供一些支持（不会被 import * from 导入）。
- 在实例的变量和方法开头加两个下划线（__）能有效地帮助把该变量或方法变成类的私有内容（using name mangling）。

- 把模块中相关的类和顶级函数放在一起。不像 Java ，这里无需要求自己在每个模块中只放一个类。但要确保放在同一模块中的类和顶级函数是高内聚的。
- 对类名使用驼峰式（形如 CapWords），而对模块名使用下划线分隔的小写字母（lower_with_under.py）。

命名样例

类别	公开的	内部的
Packages	lower_with_under	
Modules	lower_with_under	_lower_with_under
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	firstLowerCapWords()	_firstLowerCapWords()
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 *	firstLowerCapWords()	_firstLowerCapWords() (protected) or __firstLowerCapWords() (private)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

* 考虑只在首选项设置中使用对公开属性的直接访问来作为 getters 和 setters，因为函数调用在 Python 中是比较昂贵的，而 property 之后可以用来把属性访问转化为函数调用而无需改变访问语法。

3.16 程序入口

所有的 Python 源代码文件都应该是可导入的。在 Python 中，pychecker、 pydoc、 以及单元测试都要求模块是可导入的。你的代码应该总是在执行你的主程序之前检查if __name__ == '__main__': ，这样就不会在模块被导入时执行主程序。大写 main() 是故意要和命名约定的其他部分不一致，也就是说建议写成 Main() 。

```
if __name__ == '__main__':
  # 参数解析
  main()
```

[Get Code]

在模块被导入时，所有顶级缩进代码会被执行。注意不要调用函数、创建对象、或者做其他在文件被 pycheck 或 pydoc 的时候不应该做的操作。

3.17 总结

要一致

如果你在编写代码，花几分钟看看你周围的代码并且弄清它的风格。如果在 if 语句周围使用了空格，那你也应该这样做。如果注释是用星号组成的小盒子围起来的，那你同样也应该这样写。

编码风格原则的关键在于有一个编程的通用词汇表，这样人们可以人民可以集中到你要说什么而不是你要怎么说。我们在这里提供全局的编码风格规则以便人们知道这些词汇，但局部风格也重要。如果你加入文件中的代码看起来和里面已经有的代码截然不同，那在读者读它的时候就会被破坏节奏。尽量避免这样。