  2.  [Python 语言方面的准则](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc4)

  2.1  [pychecker](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide" \l "toc5)

  2.2  [导入模块和包](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc6)

  2.3  [完整路径导入](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc7)

  2.4  [异常处理](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc8)

  2.5  [全局变量](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc9)

  2.6  [内嵌/本地/内部类和函数](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc10)

  2.7  [List Comprehensions](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide" \l "toc11)

  2.8  [默认迭代器和运算符](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc12)

  2.9  [生成器](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc13)

  2.10  [使用 apply filter map reduce](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc14)

  2.11  [Lambda functions](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide" \l "toc15)

  2.12  [默认参数值](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc16)

  2.13  [Properties](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide" \l "toc17)

  2.14  [布尔内置类型](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc18)

  2.15  [String 方法](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc19)

  2.16  [静态域](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc20)

  2.17  [函数和方法修饰符](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc21)

  2.18  [线程](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc22)

  2.19  [高级特性](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#toc23)

# 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=1)

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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=2)

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](http://www.google.com/url?sa=D&q=http://pychecker.sourceforge.net)找到更多相关信息。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=6&auto=y&from=Python.PythonStyleGuide" \o "edit section:  导入模块和包 )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=3)

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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=4)

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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=5)

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=7&auto=y&from=Python.PythonStyleGuide" \o "edit section:  完整路径导入 )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=6)

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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=7)

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 。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=8&auto=y&from=Python.PythonStyleGuide" \o "edit section:  异常处理 )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=8)

* 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。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=9&auto=y&from=Python.PythonStyleGuide" \o "edit section:  全局变量 )

## 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](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide" \l "Naming)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**?**](http://www.elias.cn/Python/Naming?action=edit)章节。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=10&auto=y&from=Python.PythonStyleGuide" \o "edit section:  内嵌/本地/内部类和函数 )

## 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。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=11&auto=y&from=Python.PythonStyleGuide" \o "edit section:  List Comprehensions )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=9)

**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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=10)

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=12&auto=y&from=Python.PythonStyleGuide" \o "edit section:  默认迭代器和运算符 )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=11)

**No:**

**for** key **in** adict.keys(): ...  
  
**if** **not** adict.has\_key(key): ...  
  
**for** line **in** afile.readlines(): ...

[[$[Get Code]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=12)

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=13&auto=y&from=Python.PythonStyleGuide" \o "edit section:  生成器 )

## 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:" 。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=14&auto=y&from=Python.PythonStyleGuide" \o "edit section:  使用 apply filter map reduce )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=13)

**list comprehensions:**

[x[1] **for** x **in** my\_list **if** x[2] == 5]

[[$[Get Code]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=14)

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=15&auto=y&from=Python.PythonStyleGuide" \o "edit section:  Lambda functions )

## 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。）

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=16&auto=y&from=Python.PythonStyleGuide" \o "edit section:  默认参数值 )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=15)

**No:**

**def** foo(a, b=[]):  
  ...

[[$[Get Code]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=16)

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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=17)

**Yes:**

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

[[$[Get Code]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=18)

**No:**

foo(1, 2)

[[$[Get Code]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=19)

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=17&auto=y&from=Python.PythonStyleGuide" \o "edit section:  Properties )

## 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**?**](http://www.elias.cn/Python/FunctionAndMethodDecorators?action=edit)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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=20)

== 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。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=18&auto=y&from=Python.PythonStyleGuide" \o "edit section:  布尔内置类型 )

## 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。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=19&auto=y&from=Python.PythonStyleGuide" \o "edit section:  String 方法 )

## 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(':')

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=20&auto=y&from=Python.PythonStyleGuide" \o "edit section:  静态域 )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=21)

**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。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=21&auto=y&from=Python.PythonStyleGuide" \o "edit section:  函数和方法修饰符 )

## 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]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=22)

和如下代码是等价的：

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

[[$[Get Code]]](http://www.elias.cn/Python/PythonStyleGuide?action=sourceblock&num=23)

**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](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide" \l "Main)章节的更多讨论。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=22&auto=y&from=Python.PythonStyleGuide" \o "edit section:  线程 )

## 2.18  线程

[Google App Engine 不支持线程](http://code.google.com/appengine/kb/general.html#libraries)，所以在 SoC framework 和 Melange Web 应用程序中也别用它。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=23&auto=y&from=Python.PythonStyleGuide" \o "edit section:  高级特性 )

## 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](http://code.google.com/p/soc/wiki/MailingListsGuidelines#Developer_list)中讨论。

[(Edit)](http://www.elias.cn/Python/PythonStyleGuide?action=edit&s=24&auto=y&from=Python.PythonStyleGuide" \o "edit section:  Python 编码风格方面的准则)