编码规范(适用于python)

Version: 0.1

撰 写 人（签名）： 孙庞

完 成 日 期： 2014/4/18

本文档 使 用部门： □主管领导 ■项目组

□客户（市场） ■维护人员 □用户

评审负责人（签名）：

目录

[撰 写 人（签名）： 孙庞 1](#_Toc385597852)

[评审负责人（签名）： 1](#_Toc385597853)

[**1. 简介** 3](#_Toc385597854)

[**2. 原则** 3](#_Toc385597855)

[**3. 代码组织与风格** 3](#_Toc385597856)

[**4. 注释** 5](#_Toc385597857)

[**5. 命名** 7](#_Toc385597858)

[避免 7](#_Toc385597859)

[约定 7](#_Toc385597860)

[**6. 类** 8](#_Toc385597861)

[**7. Properties** 8](#_Toc385597862)

[**8. 异常处理** 10](#_Toc385597863)

[**9. 全局变量** 11](#_Toc385597864)

**1. 简介**

本规范为一套编写高效可靠的Python代码的标准、约定和指南。它以安全可靠的软件工程原则为基础，使代码易于理解、维护和增强，提高生产效率。同时，将带来更大的一致性，使软件开发团队的效率明显提高。

**2. 原则**

1. [导入模块和包](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Module_and_package_imports): 可以，**但不要 import \***
2. [完整路径导入](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Packages): 可以
3. [异常处理](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Exceptions): 可以
4. [全局变量](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Global_variables): 谨慎使用
5. [内嵌/本地/内部类和函数](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Nested_Local_Inner_Classes_and_Functions): 可以
6. [默认迭代器和运算符](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Default_Iterators_and_Operators): 可以
7. [生成器](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Generators): 可以
8. [使用 apply、 filter、 map、 reduce](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Using_apply_filter_map_reduce): 对**one-liner**来说可以
9. [Lambda 函数](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Lambda_functions): 对**one-liner**来说可以
10. [默认参数值](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Default_Argument_Values): 可以
11. [Properties](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Properties): 可以
12. [True/False 求值](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#True_False_evaluations): 可以
13. [布尔内置类型](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Boolean_built_in_type): 可以
14. [String 方法](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#String_Methods): 可以
15. [静态域](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Lexical_Scoping): 可以
16. [函数和方法修饰符](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Function_and_Method_Decorators): **适度**使用
17. [高级特性](http://www.elias.cn/Python/PythonStyleGuide?from=Develop.PythonStyleGuide#Power_features): **不要用**

**3. 代码组织与风格**

**3.1. 缩进**

**使用4个空格的**Tab进行缩进。

**3.2. 空行**

**建议**适当的增加空行，来增加代码的可读性。

在在类，接口以及彼此之间**要**有两行空行：

在下列情况之间**要**有一行空行：

方法之间；

局部变量和它后边的语句之间；

方法内的功能逻辑部分之间；

**3.3. 函数长度**

´ 每个函数有效代码（不包括注释和空行）长度**不要**超过60行。

**3.4. 行宽**

´ 每行代码和注释**不要**超过70个字符或屏幕的宽度，如超过则应换行，换行后的代码应该缩进一个Tab。

**3.5. 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

**No：**

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

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)

**4. 注释**

**4.1. 注释的基本约定**

注释应该增加代码的清晰度；

保持注释的简洁，不是任何代码都需要注释的，过多的注释反而会影响代码的可读性。

´ 注释**不要**包括其他的特殊字符。

**4.2. 注释类型**

**4.2.1. Doc Strings**

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

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** public\_method(self):  
    """执行一些操作。"""  
    **pass**

**4.3. 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*

**5. 命名**

**5.1. 命名的基本约定**

### 避免

* 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）。

### 约定

* 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](http://en.wikipedia.org/wiki/Cohesion_computer_science)。
* 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）。

**6. 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**

**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**

**7. 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。

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。

**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

**8. 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**

* 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。

**9. 全局变量**

