PythonCodingRule

译稿 **Python开发编码规范**

1. Python开发编码规范

```
--- hoxide 初译 dreamingk 校对发布 040724
--- xyb 重新排版 040915
--- ZoomQuiet MoinMoin 美化 050610
```

用Python进行开发时的编码风格约定 原文:**epes:PEP 008**《Style Guide for Python Code》

1.1. 介绍

这篇文档所给出的编码约定适用于在主要的Python发布版本中组成标准库的Python 代码.请查阅相关的关于在Python的C实现中C代码风格指南的描述.

这篇文档改编自Guido最初的《Python风格指南》一文. 并从《Barry's style guide》中添加了部分内容. 在有冲突的地方，Guide的风格规则应该是符合本PEP的意图 (译注：就是当有冲突时，应以Guido风格为准) 这篇PEP也许仍然尚未完成(实际上，它可能永远不会结束).

1.2. 一致性的建议

愚蠢得使用一致性是无知的妖怪(A Foolish Consistency is the Hobgoblin of Little Minds)

```
呆板的坚持一致性是傻的没边了！
-- Zoomq
```

在这篇风格指导中的一致性是重要的. 在一个项目内的一致性更重要. 在一个模块或函数内的一致性最重要.

但最重要的是:知道何时会不一致 -- 有时只是没有实施风格指导.当出现疑惑时,

运用你的最佳判断.看看别的例子,然后决定怎样看起来更好.并且要不耻下问!

- 打破一条既定规则的两个好理由:
  1. 当应用这个规则是将导致代码可读性下降,即便对某人来说,他已经习惯于按这条规则来阅读代码了.
  2. 为了和周围的代码保持一致而打破规则(也许是历史原因)
     - -- 虽然这也是个清除其它混乱的好机会(真正的XP风格).

1.3. 代码的布局

(Code lay-out)

1.3.1. 缩进

(Indentation)

使用Emacs的Python-mode的默认值:4个空格一个缩进层次. 对于确实古老的代码,你不希望产生混乱,可以继续使用8空格的制表符(8-space tabs). Emacs Python-mode自动发现文件中主要的缩进层次,依此设定缩进参数.

1.3.2. 制表符还是空格?

(Tabs or Spaces)

永远不要混用制表符和空格. 最流行的Python缩进方式是仅使用空格, 其次是仅使用制表符.混合着制表符和空格缩进的代码将被转换成仅使用空格. (在Emacs中,选中整个缓冲区,按ESC-x去除制表符(untabify).) 调用python命令行解释器时使用-t选项,可对代码中不合法得混合制表符和空格发出警告(warnings). 使用-tt时警告(warnings)将变成错误(errors).这些选项是被高度推荐的.

对于新的项目,强烈推荐仅使用空格(spaces-only)而不是制表符. 许多编辑器拥有使之易于实现的功能.(在Emacs中,确认indent-tabs-mode是nil).

1.3.3. 行的最大长度

(Maximum Line Length)

周围仍然有许多设备被限制在每行80字符;而且,窗口限制在80个字符 使将多个窗口并排放置成为可能.在这些设备上使用默认的折叠(wrapping)方式看起来有点丑陋. 因此,请将所有行限制在最大79字符(Emacs准确得将行限制为长80字符), 对顺序排放的大块文本(文档字符串或注释),推荐将长度限制在72字符.

折叠长行的首选方法是使用Pyhon支持的圆括号,方括号(brackets)和花括号(braces)内的行延续. 如果需要,你可以在表达式周围增加一对额外的圆括号, 但是有时使用反斜杠看起来更好.确认恰当得缩进了延续的行. Emacs的Python-mode正确得完成了这些.一些例子:

Toggle line numbers

```
1         class Rectangle(Blob):
2
3                 def __init__(self, width, height,
4                                            color='black', emphasis=None, highlight=0):
5                         if width == 0 and height == 0 and \
                        color == 'red' and emphasis == 'strong' or \
                        highlight > 100:
6                                 raise ValueError, "sorry, you lose"
7                         if width == 0 and height == 0 and (color == 'red' or
8                                                                                 emphasis is None):
9                                 raise ValueError, "I don't think so"
10                        Blob.__init__(self, width, height,
11                                                    color, emphasis, highlight)
```

1.3.4. 空行

(Blank Lines)

用两行空行分割顶层函数和类的定义,类内方法的定义用单个空行分割. 额外的空行可被用于(保守的(sparingly))分割相关函数组成的群(groups of related functions). 在一组相关的单句中间可以省略空行.(例如.一组哑元(a set of dummy implementations)).

当空行用于分割方法(method)的定义时,在'class'行和第一个方法定义之间也要有一个空行.

在函数中使用空行时,请谨慎的用于表示一个逻辑段落(indicate logical sections).

Python接受contol-L(即^L)换页符作为空格;Emacs(和一些打印工具) 视这个字符为页面分割符,因此在你的文件中,可以用他们来为相关片段(sections)分页.

1.3.5. 编码

(Encodings)epes:(PEP 263)

Python核心发布中的代码必须始终使用ASCII或Latin-1编码(又名 ISO-8859-1). 使用ASCII的文件不必有译码cookie(coding cookie). Latin-1仅当注释或文档字符串涉及作者名字需要Latin-1时才被使用; 另外使用\x转义字符是在字符串中包含非ASCII(non-ASCII)数据

的首选方法. 作为PEP 263实现代码的测试套件的部分文件是个例外.

```
Python 2.4 以后内核支持 Unicode 了!
不论什么情况使用 UTF-8 吧! 这是王道!
```

--ZoomQuiet

1.4. 导入

(Imports)

- 通常应该在单独的行中导入(Imports),例如:

```
No:  import sys, os
Yes: import sys
         import os
```

但是这样也是可以的:

```
from types import StringType, ListType
```

- Imports 通常被放置在文件的顶部,仅在模块注释和文档字符串之后,在模块的全局变量和常量之前.Imports应该有顺序地成组安放.
  1. 标准库的导入(Imports)
  2. 相关的主包(major package)的导入(即,所有的email包在随后导入)
  3. 特定应用的导入(imports)
- 你应该在每组导入之间放置一个空行.
- 对于内部包的导入是不推荐使用相对导入的.对所有导入都要使用包的绝对路径.
- 从一个包含类的模块中导入类时,通常可以写成这样:

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

如果这样写导致了本地名字冲突,那么就这样写

```
import MyClass
```

```
import foo.bar.YourClass
```

- 即使用"MyClass.MyClass"和"foo.bar.YourClass.YourClass"

1.5. 空格

(Whitespace in Expressions and Statements)

Guido不喜欢在以下地方出现空格:

"spam(ham[1], { eggs: 2 })". Always write this as "spam(ham[1], {eggs: 2})".

- 紧挨着圆括号,方括号和花括号的,如:"spam(ham[1], { eggs: 2 })".

要始终将它写成"spam(ham[1], {eggs: 2})".

"if x == 4 : print x , y ; x , y = y , x". Always write this as "if x == 4: print x, y; x, y = y, x".

- 紧贴在逗号,分号或冒号前的,如:

"if x == 4 : print x , y ; x , y = y , x". 要始终将它写成"if x == 4: print x, y; x, y = y, x".

- 紧贴着函数调用的参数列表前开式括号(open parenthesis)的,如"spam (1)".要始终将它写成"spam(1)".

slicing, as in: "dict ['key'] = list [index]". Always write this as "dict['key'] = list[index]".

- 紧贴在索引或切片(slicing?下标?)开始的开式括号前的,如:

"dict ['key'] = list [index]".要始终将它写成"dict['key'] = list[index]".

- 在赋值(或其它)运算符周围的用于和其它并排的一个以上的空格,如:

Toggle line numbers

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

要始终将它写成

Toggle line numbers

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

(不要对以上任意一条和他争论 --- Guido 养成这样的风格超过20年了.)

1.5.1. 其它建议

(Other Recommendations)

- 始终在这些二元运算符两边放置一个空格:赋值(=), 比较(==, <, >, !=, <>, <=,>=, in, not in, is, is not), 布尔运算 (and, or, not).

* 按你的看法在算术运算符周围插入空格. 始终保持二元运算符两边空格的一致.

- 一些例子:

Toggle line numbers

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

- 不要在用于指定关键字参数或默认参数值的'='号周围使用空格,例如:

Toggle line numbers

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

- 不要将多条语句写在同一行上.

```
                No:  if foo == 'blah': do_blah_thing()
                Yes: if foo == 'blah':
                         do_blah_thing()

                No:  do_one(); do_two(); do_three()
                Yes: do_one()
                     do_two()
                     do_three()
```

1.6. 注释

(Comments)

同代码不一致的注释比没注释更差.当代码修改时,始终优先更新注释!

注释应该是完整的句子. 如果注释是一个短语或句子,首字母应该大写, 除非他是一个以小写字母开头的标识符(永远不要修改标识符的大小写).

如果注释很短,最好省略末尾的句号(period?结尾句末的停顿?也可以是逗号吧,) 注释块通常由一个或多个由完整句子构成的段落组成,每个句子应该以句号结尾.

你应该在句末,句号后使用两个空格,以便使Emacs的断行和填充工作协调一致 (译按:应该说是使这两种功能正常工作,". "给出了文档结构的提示).

用英语书写时,断词和空格是可用的.

非英语国家的Python程序员:请用英语书写你的注释,除非你120%的确信 这些代码不会被不懂你的语言的人阅读.

> 我就是坚持全部使用中文来注释，真正要发布脚本工具时，再想E文的；
> 开发时每一瞬间都要用在思量中，坚决不用在E文语法，单词的回忆中！

-- ZoomQUiet

- 约定使用统一的文档化注释格式有利于良好习惯和团队建议！ -- CodeCommentingRule

1.6.1. 注释块

(Block Comments)

注释块通常应用于跟随着一些(或者全部)代码并和这些代码有着相同的缩进层次. 注释块中每行以'#'和一个空格开始(除非他是注释内的缩进文本). 注释块内的段落以仅含单个'#'的行分割. 注释块上下方最好有一空行包围(或上方两行下方一行,对一个新函数定义段的注释).

1.6.2. 行内注释

(Inline Comments)

- (inline?内联?翻成"行内"比较好吧)

  一个行内注释是和语句在同一行的注释.行内注释应该谨慎适用. 行内注释应该至少用两个空格和语句分开. 它们应该以'#'和单个空格开始.

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

如果语意是很明了的,那么行内注释是不必要的,事实上是应该被去掉的. 不要这样写:

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

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

但是有时,这样是有益的:

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

1.7. 文档化

(Documentation Strings)

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in epes:PEP 257.

应该一直遵守编写好的文档字符串(又名"docstrings")的约定(?实在不知道怎么译)

```
Documentation Strings-- 文档化字符 ;
为配合 pydoc;epydoc,Doxygen等等文档化工具的使用,类似于MoinMoin 语法,约定一些字符,
以便自动提取转化为有意义的文档章节等等文章元素!
-- Zoomq
```

- 为所有公共模块,函数,类和方法编写文档字符串.文档字符串对非公开的方法不是必要的,但你应该有一个描述这个方法做什么的注释.这个注释应该在"def"这行后.
- epes:PEP 257 描述了好的文档字符串的约定.一定注意,多行文档字符串结尾的""" 应该单独成行,例如:

```
        """Return a foobang

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

- 对单行的文档字符串,结尾的"""在同一行也可以.

```
实际上Python 自个儿就使用文档化编码维护着所有内置对象的使用说明\
不信的话常试：
        #python
>>> import time
>>> dir(time)
['__doc__', '__file__', '__name__', 'accept2dyear', 'altzone', 'asctime', 'clock', 'ctime', 'daylight', 'gmtime',
'localtime', 'mktime', 'sleep', 'strftime', 'strptime', 'struct_time', 'time', 'timezone', 'tzname', 'tzset']
>>> help(time.time)
Help on built-in function time in module time:

time(...)
        time() -> floating point number

        Return the current time in seconds since the Epoch.
        Fractions of a second may be present if the system clock provides them.
```

1.8. 版本注记

(Version Bookkeeping) (我觉得叫"注记"更好)

如果你要将RCS或CVS的杂项(crud)包含在你的源文件中,按如下做.

Toggle line numbers

```
1                   __version__ = "$Revision: 1.4 $"
2                   # $Source: E:/cvsroot/python_doc/pep8.txt,v $
```

这个行应该包含在模块的文档字符串之后,所有代码之前,上下用一个空行分割.

```
对于CVS的服务器工作标记更应该在代码段中明确出它的使用
如：在文档的最开始的版权声明后应加入如下版本标记：
# 文件：$id$
# 版本：  $Revision$
这样的标记在提交给配置管理服务器后，会自动适配成为相应的字符串，如：
# 文件：$Id: ussp.py,v 1.22 2004/07/21 04:47:41 hd Exp $
```

```
# 版本:  $Revision: 1.4 $
----HD
```

1.9. 命名约定

(Naming Conventions)

Python库的命名约定有点混乱,所以我们将永远不能使之变得完全一致--- 不过还是有公认的命名规范的. 新的模块和包(包括第三方的框架)必须符合这些标准,但对已有的库存在不同风格的, 保持内部的一致性是首选的.

1.9.1. 描述:命名风格

(Descriptive: Naming Styles)

有许多不同的命名风格.以下的有助于辨认正在使用的命名风格,独立于它们的作用.

以下的命名风格是众所周知的:

- b (单个小写字母)
- B (单个大写字母)
- 小写串 如:getname
- 带下划的小写串 如:_getname
- 大写串 如:GETNAME
- 带下划的大写串 如:_GETNAME
- CapitalizedWords(首字母大写单词串) (或 CapWords, CamelCase -- 这样命名是由于它的字母错落有致的样子而来的.

  这有时也被当作StudlyCaps. 如:GetName
- mixedCase (混合大小写串)(与首字母大写串不同之处在于第一个字符是小写如:getName)
- Capitalized_Words_With_Underscores(带下划线的首字母大写串) (丑陋!)

  还有一种使用特别前缀的风格，用于将相关的名字分成组.这在Python中不常用, 但是出于完整性要提一下.例如,os.stat()函数返回一个tuple, 他的元素传统上有象st_mode, st_size, st_mtime等等这样的名字. X11库的所有公开函数以X开头.(在Python中,这个风格通常认为是不必要的, 因为属性和方法名以对象作前缀,而函数名以模块名作前缀.)

  另外,以下用下划线作前导或结尾的特殊形式是被公认的(这些通常可以和任何习惯组合(使用?)):

- _single_leading_underscore(以一个下划线作前导): 弱的"内部使用(internal use)"标志.
  - (例如,"from M import *"不会导入以下划线开头的对象).

- single_trailing_underscore_(以一个下划线结尾): 用于避免与Python关键词的冲突,例如.
  - "Tkinter.Toplevel(master, class_='ClassName')".
- __double_leading_underscore(双下划线): 从Python 1.4起为类私有名.
- __double_leading_and_trailing_underscore__: 特殊的(magic) 对象或属性,存在于用户控制的(user-controlled)名字空间, 例如: __init__, __import__ 或 __file__. 有时它们被用户定义, 用于触发某个特殊行为(magic behavior)(例如:运算符重载); 有时被构造器(infrastructure)插入,以便自己使用或为了调试. 因此,在未来的版本中,构造器(松散得定义为Python解释器和标准库) 可能打算建立自己的魔法属性列表,用户代码通常应该限制将这种约定作为己用. 欲成为构造器的一部分的用户代码可以在下滑线中结合使用短前缀,例如. __bobo_magic_attr__.

1.9.2. 说明:命名约定

(Prescriptive: Naming Conventions)

1.9.2.1. 应避免的名字

(Names to Avoid)

永远不要用字符'l'(小写字母el(就是读音,下同)), O (大写字母oh), 或I'(大写字母eye)作为单字符的变量名. 在某些字体中,这些字符不能与数字1和0分开.当想要使用'l'时，用'L'代替它.

1.9.2.2. 模块名

(Module Names)

模块应该是不含下划线的,简短的,小写的名字.

因为模块名被映射到文件名, 有些文件系统大小写不敏感并且截短长名字, 模块名被选为相当短是重要的---这在Unix上不是问题, 但当代码传到Mac 或Windows上就可能是个问题了.

当一个用C或C++写的扩展模块有一个伴随的Python模块,这个Python模块提供了

一个更高层(例如，更面向对象)的接口时,C/C++模块有一个前导下划线(如：_socket)

Python包应该是不含下划线的,简短的,全小写的名字.

1.9.2.3. 类名

(Class Names)

几乎没有例外，类名总是使用首字母大写单词串(CapWords)的约定.

1.9.2.4. 异常名

(Exception Names)

如果模块对所有情况定义了单个异常,它通常被叫做"error"或"Error". 似乎内建(扩展)的模块使用"error"(例如:os.error), 而Python模块通常用"Error" (例如: xdrlib.Error). 趋势似乎是倾向使用CapWords异常名.

1.9.2.5. 全局变量名

(Global Variable Names)

(让我们希望这些变量打算只被用于模块内部) 这些约定与那些用于函数的约定差不多.被设计可以通过"from M import *"来使用的那些模块,应该在那些不想被导入的全局变量(还有内部函数和类)前加一个下划线).

1.9.2.6. 函数名

(Function Names)

函数名应该为小写,可能用下划线风格单词以增加可读性. mixedCase仅被允许用于这种风格已经占优势的上下文(如: threading.py) 以便保持向后兼容.

1.9.2.7. 方法名和实例变量

(Method Names and Instance Variables)

这段大体上和函数相同:通常使用小写单词,必要时用下划线分隔增加可读性.

使用一个前导下划线仅用于不打算作为类的公共接口的内部方法和实例变量. Python不强制要求这样; 它取决于程序员是否遵守这个约定.

使用两个前导下划线以表示类私有的名字. Python将这些名字和类名连接在一起: 如果类Foo有一个属性名为 __a, 它不能以Foo. __a访问. (执著的用户(An insistent user)还是可以通过Foo. _Foo__a得到访问权.) 通常,双前导下划线应该只用来避免与类(为可以子类化所设

计)中的属性发生名字冲突.

1.9.2.8. 继承的设计

(Designing for inheritance)

始终要确定一个类中的方法和实例变量是否要被公开. 通常,永远不要将数据变量公开,除非你实现的本质上只是记录. 人们总是更喜欢给类提供一个函数的接口作为替换 (Python 2.2 的一些开发者在这点上做得非常漂亮).

同样,确定你的属性是否应为私有的.私有与非公有的区别在于: 前者永远不会被用在一个派生类中,而后者可能会. 是的,你应该在大脑中就用继承设计好了你的类.

私有属性必须有两个前导下划线,无后置下划线.

非公有属性必须有一个前导下划线,无后置下划线.

公共属性没有前导和后置下划线,除非它们与保留字冲突, 在此情况下,单个后置下划线比前置或混乱的拼写要好, 例如:class_优于klass. 最后一点有些争议; 如果相比class_你更喜欢klass,那么这只是一致性问题.

1.10. 设计建议

(Programming Recommendations)

- 同象None之类的单值进行比较,应该永远用:'is'或'is not'来做. 当你本意是"if x is not None"时,对写成"if x"要小心 -- 例如当你测试一个默认为None的变量或参数是否被设置为其它值时. 这个其它值可能是一个在布尔上下文中为假的值!
- 基于类的异常总是好过基于字符串的异常. 模块和包应该定义它们自己的域内特定的基异常类(base exception class), 基类应该是内建的Exception类的子类. 还始终包含一个类的文档字符串.例如:

Toggle line numbers

```
1                class MessageError(Exception):
2                        """Base class for errors in the email package."""
```

- 使用字符串方法(methods)代替字符串模块,除非必须向后兼容Python 2.0以前的版本. 字符串方法总是非常快,而且和unicode字符串共用同样的API(应用程序接口)
- 在检查前缀或后缀时避免对字符串进行切片.

  用startswith()和endswith()代替, 因为它们是明确的并且错误更少. 例如:

```
No:  if foo[:3] == 'bar':
```

```
Yes: if foo.startswith('bar'):
```

例外是如果你的代码必须工作在Python 1.5.2 (但是我们希望它不会发生!).

- 对象类型的比较应该始终用isinstance()代替直接比较类型.例如:

```
No:  if type(obj) is type(1):
Yes: if isinstance(obj, int):
```

检查一个对象是否是字符串时,紧记它也可能是unicode字符串! 在Python 2.3, str和unicode有公共的基类,basestring,所以你可以这样做:

Toggle line numbers

```
1 if isinstance(obj, basestring):
```

在Python 2.2 类型模块为此定义了StringTypes类型, 例如:

Toggle line numbers

```
1 from types import StringTypes
2 if isinstance(obj, StringTypes):
```

在Python 2.0和2.1,你应该这样做:

Toggle line numbers

```
1 from types import StringType, UnicodeType
2 if isinstance(obj, StringType) or \
  isinstance(obj, UnicodeType) :
```

- 对序列,(字符串(strings),列表(lists),元组(tuples)), 使用空列表是false这个事实,因此"if not seq"或"if seq"比 "if len(seq)"或"if not len(seq)"好.
- 书写字符串文字时不要依赖于有意义的后置空格. 这种后置空格在视觉上是不可辨别的,并且有些编辑器(特别是近来,reindent.py) 会将它们修整掉.
- 不要用 == 来比较布尔型的值以确定是True或False(布尔型是Pythn 2.3中新增的)

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

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

-- ZoomQuiet (2005-01-26)

last edited 2005-06-17 02:48:56 by ZoomQuiet