Python 开发编码规范

--- hoxide 初译
dreamingk 校对发布 040724
--- xyb 重新排版 040915

RedWolf pdf 整理 080807

PEP: 8

Title: Style Guide for Python Code

Version: 63990

Last-Modified:2008-06-06 20:48:49 +0200 (Fri, 06 Jun 2008)

Author: Guido van Rossum <guido at python.org>,
Barry Warsaw <barry at python.org>

Status:Active

Type: Process

Created: 05-Jul-2001

Post-History:05-Jul-2001

网址: http://wiki.woodpecker.org.cn/moin/Python 开发编码规范
http://www.python.org/dev/peps/pep-0008/

目　　录

用 python 进行开发时的编码风格约定

1. 介绍(Introduction)

- This document gives coding conventions for the Python code comprising the standard library for the main Python describing style guidelines for the C code in the C implementation of Python[1].

  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.

- This document was adapted from Guido's original Python Style Guide essay[2], with some additions from Barry's style guide[5]. Where there's conflict, Guido's style rules for the purposes of this PEP. This PEP may still be incomplete (in fact, it may never be finished <wink>).

  这篇文档改编自 Guido 最初的《Python 风格指南》一文. 并从《Barry's style guide》[5]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).

2. 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

- One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 [6] says, "Readability counts".

  Guido 有一个重要的观点是：代码的阅读比编写更广泛。因此，一个准则

是,有计划的改进代码的可读性，使得广泛的python代码有一致性，正是PEP 20 [6] 所说的“可读率”。

- A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

  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.

- But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

  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!

  Two good reasons to break a particular rule:

  打破一条既定规则的两个好理由:

  (1) When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules.

  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) To be consistent with surrounding code that also breaks it (maybe for historic reasons) -- although this is also an opportunity to clean up someone else's mess (in true XP style).

  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(眞正的XP风格).

3. 代码的布局(Code lay-out)

3.1. 缩进(Indentation)

- Use the default of Emacs' Python-mode: 4 spaces for one indentation level. For really old code that you don't want to mess up, you can continue to use 8-space tabs. Emacs Python-mode auto-detects the prevailing indentation level used in a file and sets its indentation parameters accordingly.

  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.

3.2. 制表符还是空格(Tabs or Spaces)?

- Never mix tabs and spaces.

  The most popular way of indenting Python is with spaces only. The second-most popular way is with tabs only. Code indented with a mixture of tabs and spaces should be converted to using spaces exclusively. (In Emacs, select the whole buffer and hit ESC-x untabify.) When invoking the python command line interpreter with the -t option, it issues warnings about code that illegally mixes tabs and spaces. When using -tt these warnings become errors. These options are highly recommended!

  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.

- For new projects, spaces-only are strongly recommended over tabs. Most editors have features that make this easy to do. (In Emacs, make sure indent-tabs-mode is nil).

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).

3.3. 行的最大长度(Maximum Line Length)

- There are still many devices around that are limited to 80 character lines; plus, limiting windows to 80 characters makes it possible to have several windows side-by-side. The default wrapping on such devices looks ugly. Therefore, please limit all lines to a maximum of 79 characters (Emacs wraps lines that are exactly 80 characters long). For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

  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.

- The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. If necessary, you can add an extra pair of parentheses around an expression, but sometimes using a backslash looks better. Make sure to indent the continued line appropriately. Emacs Python-mode does this right. Some examples:

  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:

```
class Rectangle(Blob):

    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError("sorry, you lose")
            if width == 0 and height == 0 and (color == 'red' or
                                               emphasis is None):
                raise ValueError("I don't think so -- values are %s, %s" %
                                 (width, height))
            Blob.__init__(self, width, height,
                          color, emphasis, highlight)
```

3.4. 空行(Blank Lines)

- Separate top-level function and class definitions with two blank lines. Method definitions inside a class are separated by a single blank line. Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (e.g. a set of dummy implementations).

  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)).

- When blank lines are used to separate method definitions, there is also a blank line between the 'class' line and the first method definition.

  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.

- Use blank lines in functions, sparingly, to indicate logical sections.

- 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 accepts the control-L (i.e. ^L) form feed character as whitespace; Emacs (and some printing tools) treat these characters as page separators, so you may use them to separate pages of related sections of your file.

  Python 接受 conto1-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.

3.5. 编码(Encodings)(PEP 263)

- Code in the core Python distribution should aways use the ASCII or Latin-1 encoding (a.k.a. ISO-8859-1). For Python 3.0 and beyond,UTF-8 is preferred over Latin-1, see PEP 3120. Files using ASCII(or UTF-8, for Python 3.0) should not have a coding cookie. Latin-1(or UTF-8) should only be used when a comment or docstring needs to mention an author name that requires Latin-1; otherwise, using \x \u or \U escapes is the preferred way to include non-ASCII data in string literals.

  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). Python3.0 或更高的版本，UTF-8 优先于 Latin-1 ，可查看 PEP 3120.使用 ASCII(Python3.0 中的 UTF-8)的文件不必有译码 cookie(coding cookie). Latin-1(或 UTF-8)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 \u 或\U 转义字符是在字符串中包含非 ASCII(non-ASCII)数据的首选方法.
- For Python 3.0 and beyond, the following policy is prescribed for the standard library (see PEP 3131): All identifiers in the Python standard library MUST use ASCII-only identifiers, and SHOULD use English words wherever feasible (in many cases, abbreviations and technical terms are used which aren't English). In addition, string literals and comments must also be in ASCII. The only exceptions are test cases testing the non-ASCII features, and names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin

transliteration of their names.

对于 python3.0 或更高版本,关于标准库有以下规定(查看 PEP 3131):Python 标准库中的所有标志符必须并且只能使用 ASII 标志符，应当使用恰当的英语词汇（在很多场合,一些缩写和专业术语并不是英语).另外,字母字符串和注释也必须使用 ASCII 标志符.只有当测试非 ASII 特征和书写著者的姓名才是例外的.那些不是基于拉丁字母的著者姓名必须提供一个拉丁字母的名字！！

- Open source projects with a global audience are encouraged to adopt a similar policy.
  全球参与开源项目的同志,鼓励采用同类的约定！

4. 导入(Imports)

- Imports should usually be on separate lines, e.g.:

  通常应该在单独的行中导入(Imports),例如:

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

it's okay to say this though:

但是这样也是可以的:

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

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants. Imports should be grouped, with the order being

  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. standard library Imports
     标准库的导入(Imports)
  2. related major package imports (i.e. all email package imports next)
     相关的主包(major package)的导入(即,所有的 email 包在随后导入)

3. application specific imports

特定应用的导入(imports) You should put a blank line between each group of imports. 你应该在每组导入之间放置一个空行.

- Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports.
- 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.
- When importing a class from a class-containing module, it's usually okay to spell this

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
```

If this spelling causes local name clashes, then spell them

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
```

and use "MyClass.MyClass" and "foo.bar.YourClass.YourClass"

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

5. 表达式和语句中的空格 (Whitespace in Expressions and Statements)

Pet Peeves

Avoid extraneous whitespace in the following situations:

避免以下集中不必要的空格:

- Immediately inside parentheses, brackets or braces, as in:

"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})".

- Immediately before a comma, semicolon, or colon, as in:

  "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".

- Immediately before the open parenthesis that starts the argument list of a function call, as in "spam (1)". Always write this as "spam(1)".

  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)".

- Immediately before the open parenthesis that starts an indexing or 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]".

- More than one space around an assignment (or other) operator to align it with another, as in:
  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:

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

Always write this as
要始终将它写成

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

(Don't bother to argue with him on any of the above -- Guido's grown accustomed to this style over 20 years.)
(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.)

5.1. 其它建议(Other Recommendations)

- Always surround these binary operators with a single space on
- either side: assignment (=), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

  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).
- Use your better judgment for the insertion of spaces around arithmetic operators. Always be consistent about whitespace on either side of a binary operator. Some examples:
  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:

```
Yes:
        i = i + 1
        submitted += 1
        x = x * 2 - 1
        hypot2 = x * x + y * y
        c = (a + b) * (a — b)
```

```
No:
    i=i+1
    submitted +=1
    x = x*2 - 1
    hypot2 = x*x + y*y
    c = (a+b) * (a-b)
```

Don't use spaces around the '=' sign when used to indicate a keyword argument or a default parameter value. For instance:

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:

```
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)
```

Compound statements (multiple statements on the same line) are generally discouraged.

不要将多条语句写在同一行上.

```
Yes:
    if foo == 'blah':
        do_blah_thing()
    do_one()
    do_two()
    do_three()

Rather not:
    if foo == 'blah': do_blah_thing()
    do_one(); do_two(); do_three()
```

While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements.

Also avoid folding such long lines!

尽管有时在 if/for/while 中可以把较短的代码写在同一行,但绝不要把多于句写在同一行.避免将语句折叠成长行！

```
Rather not:
     if foo == 'blah': do_blah_thing()
     for x in lst: total += x
     while t < 10: t = delay()

 Definitely not:
     if foo == 'blah': do_blah_thing()
     else: do_non_blah_thing()

     try: something()
     finally: cleanup()

     do_one(); do_two(); do_three(long, argument,
                                  list, like, this)

     if foo == 'blah': one(); two(); three()
```

6. 注释(Comments)

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

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!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

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).

If a comment is short, the period at the end is best omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

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.

You should use two spaces after a sentence-ending period, since it makes Emacs wrapping and filling work consistenty.

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).

When writing English, Strunk and White apply.

用英语书写时,断词和空格是可用的.

Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

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.

6.1. 注释块(Block Comments)

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line containing a single #. Block comments are best surrounded by a blank line above and below them (or two lines above and a single line below for a block comment at the start of a a new section of function definitions).
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).

6.2. 行内注释(Inline Comments)

(inline?内联?翻成"行内"比较好吧)

- An inline comment is a comment on the same line as a statement. Inline comments should be used sparingly. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

  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.

- Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

  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
```

But sometimes, this is useful:

但是有时,这样是有益的:

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

7. 文档字符串(Documentation Strings)

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in PEP 257 [3].
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)PEP 257 [3].

Documentation Strings-- 文档化字符 ;
为了 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

- Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for non-public methods, but you should have a comment that describes what the method does. This comment should appear after the "def" line.

  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.
- PEP 257 describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself, e.g.:
  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.
 """
```

- For one liner docstrings, it's okay to keep the closing """ on the same line.

  对单行的文档字符串,结尾的"""在同一行也可以.

8. 版本注记(Version Bookkeeping)

- If you have to have RCS or CVS crud in your source file, do it as follows.

  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.

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

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.
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

9. 命名约定(Naming Conventions)

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent -- nevertheless, here are the currently recommended naming standards. New modules and packages (including 3rd party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.
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.

9.1. 描述:命名风格(Descriptive: Naming Styles)

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

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.

The following naming styles are commonly distinguished:

以下的命名风格是众所周知的:

- b (single lowercase letter)
  b (单个小写字母)
- B (single uppercase letter)
  B (单个大写字母)
- lowercase
  小写串 如:getname
- lower_case_with_underscores
  带下划的小写串 如:_getname
- UPPERCASE
  大写串 如:GETNAME
- UPPER_CASE_WITH_UNDERSCORES
  带下划的大写串 如:_GETNAME
- CapitalizedWords(or CapWords, or CamelCase -- so named because of the bumpy look of its letters[4]). This is also sometimes known as StudlyCpas.

  CapitalizedWords(首字母大写单词串)(或CapWords, 或 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StudlyCpas. 如:GetName
- mixedCase (differs from CapitalizedWords by initial lowercase character!)

  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 (ugly!)
  Capitalized_Words_With_Underscores(带下划线的首字母大写串)(丑陋!)
- There's also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the os.stat() function returns a tuple whose items traditionally have names like st_mode, st_size, st_mtime and so on. The X11 library uses

a leading X for all its public functions. (In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.)
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.)

- In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):
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: weak "internal use" indicator (e.g. "from M import *" does not import objects whose name starts with an underscore).
_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_: used by convention to avoid conflicts with Python keyword, e.g.

  "Tkinter.Toplevel(master, class_='ClassName')".
- 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: class-private names as of Python 1.4.

  __double_leading_underscore(双下划线): 从Python 1.4起为类私有名.
- __double_leading_and_trailing_underscore__: "magic" objects or

attributes that live in user-controlled namespaces, e.g. __init__, __import__ or __file__. Sometimes these are defined by the user to trigger certain magic behavior (e.g. operator overloading); sometimes these are inserted by the infrastructure for its own use or for debugging purposes. Since the infrastructure (loosely defined as the Python interpreter and the standard library) may decide to grow its list of magic attributes in future versions, user code should generally refrain from using this convention for its own use. User code that aspires to become part of the infrastructure could combine this with a short prefix inside the underscores, e.g. __bobo_magic_attr__.

__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__.

9.2. 说明:命名约定(Prescriptive: Naming Conventions)

9.2.1. 应避免的名字(Names to Avoid)

Never use the characters 1' (lowercase letter el), 0' (uppercase letter oh), or `I' (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguisable from the numerals one and zero. When tempted to use 1' use L' instead.

永远不要用字符`1'(小写字母 el(就是读音,下同)),

0'(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'1'时，用'L'代替它.

9.2.2. 模块名(Module Names)

Modules should have short, lowercase names, without underscores.

模块应该是不含下划线的,简短的,小写的名字.

Since module names are mapped to file names, and some file systems are case insensitive and truncate long names, it is important that module names be chosen to be fairly short -- this won't be a problem on Unix, but it may be a problem when the code is transported to Mac or Windows.

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.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. _socket).

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 packages should have short, all-lowercase names, without underscores.

Python包应该是不含下划线的,简短的,全小写的名字.

9.2.3. 类名(Class Names)

Almost without exception, class names use the CapWords convention. Classes for internal use have a leading underscore in addition.

几乎没有例外，类名总是使用首字母大写单词串(CapWords)的约定.

9.2.4. 异常名(Exception Names)

If a module defines a single exception raised for all sorts of conditions, it is generally called "error" or "Error". It seems that built-in (extension) modules use "error" (e.g. os.error), while Python modules generally use "Error" (e.g. xdrlib.Error).
The trend seems to be toward CapWords 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.

9.2.5. 全局变量名(Global Variable Names)

(Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions. Modules that are designed for use via "from M import *" should prefix their globals (and internal functions and classes) with an underscore to prevent exporting them.

(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).

9.2.6. 函数名(Function Names)

Function names should be lowercase, possibly with words separated by underscores to improve readability. mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

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.

9.2.7. 方法名和实例变量 (Method Names and Instance Variables)

The story is largely the same as with functions: in general, use lowercase with words separated by underscores as necessary to improve readability.

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.

Use one leading underscore only for internal methods and instance variables which are not intended to be part of the class's public interface. Python does not enforce this; it is up to programmers to respect the convention.

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.

Use two leading underscores to denote class-private names. Python "mangles" these names with the class name: if class Foo has an attribute named __a, it cannot be accessed by Foo.__a. (An insistent user could still gain access by calling Foo._Foo__a.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

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.

9.2.8. 继承的设计(Designing for inheritance)

Always decide whether a class's methods and instance variables should be public or non-public. In general, never make data variables public unless you're implementing essentially a record. It's almost always preferrable to give a functional interface to your class instead (and some Python 2.2 developments will make this much nicer).

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).

Also decide whether your attributes should be private or not. The difference between private and non-public is that the former will never be useful for a derived class, while the latter might be. Yes, you should design your classes with inheritence in mind!

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.

Private attributes should have two leading underscores, no trailing underscores.

私有属性必须有两个前导下划线,无后置下划线.

Non-public attributes should have a single leading underscore, no trailing underscores.

非公有属性必须有一个前导下划线,无后置下划线.

Public attributes should have no leading or trailing underscores, unless they conflict with reserved words, in which case, a single trailing underscore is preferrable to a leading one, or a corrupted spelling, e.g. class_ rather than klass. (This last point is a bit controversial; if you prefer klass over class_ then just be consistent. :).

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.

10. 设计建议(Programming Recommendations)

- Comparisons to singletons like None should always be done with 'is' or 'is not'. Also, beware of writing "if x" when you really mean "if x is not None" -- e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might be a value that's false in a Boolean context!

  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!

- Class-based exceptions are always preferred over string-based exceptions. Modules or packages should define their own domain-specific base exception class, which should be subclassed from the built-in Exception class. Always include a class docstring. E.g.:

  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:

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

- Use string methods instead of the string module unless backward-compatibility with versions earlier than Python 2.0 is important. String methods are always much faster and share the same API with unicode strings.

  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)

- Avoid slicing strings when checking for prefixes or suffixes. Use startswith() and endswith() instead, since they are cleaner and less error prone. For example:

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'):
```

- The exception is if your code must work with Python 1.5.2 (but let's hope not!).

  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!).

- Object type comparisons should always use isinstance() instead of comparing types directly. E.g.
  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):
```

When checking if an object is a string, keep in mind that it might be a unicode string too! In Python 2.3, str and unicode have a common base class, basestring, so you can do:

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：

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

In Python 2.2, the types module has the StringTypes type defined for that purpose, e.g.:

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

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

In Python 2.0 and 2.1, you should do:
在 Python 2.0 和 2.1,你应该这样做：

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

```
        isinstance(obj, UnicodeType) :
```

- For sequences, (strings, lists, tuples), use the fact that empty sequences are false, so "if not seq" or "if seq" is preferable to "if len(seq)" or "if not len(seq)".

  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.
- Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently,reindent.py) will trim them.
  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.
- Don't compare boolean values to True or False using == (bool types are new in Python 2.3):
  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:
```