[**Introduction 介绍**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id9)

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python [[1]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id5).

本文档列出的编码约定，是基于主Python发行版标准库的Python代码的。Python的C语言实现的C代码风格指南，请参阅PEP 7[[1]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id5)。

This document was adapted from Guido's original Python Style Guide essay [[2]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id6), with some additions from Barry's style guide [[3]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id7). 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（译者注：Python之父，[Guido van Rossum](http://en.wikipedia.org/wiki/Guido_van_Rossum)）最初的《Python Style Guide》[[2]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id6)一文，《Barry's style guide》[[3]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id7)亦有贡献。对（2文）有冲突的地方，Guido的风格指南更符合这份PEP思想。当然，这份PEP仍不完善（实际上，它可能永远也无法完善<wink>）

[**A Foolish Consistency is the Hobgoblin of Little Minds 愚蠢的一致性就像没脑子的妖怪**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id10)

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](http://python.org/dev/peps/pep-0020) says, "Readability counts".

Guido的关键见解之一是阅读代码比怎么写代码更重要。这份指南意图提高代码的可阅读性，使整个 Python 代码的代码保持一致。正如 [PEP 20](http://python.org/dev/peps/pep-0020) 所表述的，“可读性很重要”。（译者注：[可以点击这里查看由lanphaday翻译的“The Zen of Python”](http://blog.csdn.net/lanphaday/article/details/2151918)。）

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.

这是一份关于一致性的风格指南。这份风格指南的风格一致性是非常重要的。更重要的是项目的风格一致性。在一个模块或函数的风格一致性是最重要的。

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：知道什么时候是不一致的————有时风格指南并不适用。有疑问时，请使用您的最佳判断。看看其他的例子，并决定什么看上去最美观。不要犹豫，提示提问！

Two good reasons to break a particular rule:

两个很好的理由来打破特定规则：

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

应用了这个规则后，代码的可读性却降低了。甚至有人依据以下的规则来阅读代码。

1. 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).

欲与周围的代码的代码保持一致，也会打破这个规则（也许是历史的原因）————虽然这也是一个机会来收拾别人的烂摊子（真正的XP风格）。（译者注：XP，Xtreme Programming，极限编程）

[**Code lay-out 代码布局**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id11)

[**Indentation 缩进和换行**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id12)

Use 4 spaces per indentation level.

每级缩进使用 4 个空格。

For really old code that you don't want to mess up, you can continue to use 8-space tabs.

如果不想搞乱以前的古老的代码的话，可以使用8个空格长的制表符。

Continuation lines should align wrapped elements either vertically using Python's implicit line joining inside parentheses, brackets and braces, or using a hanging indent. When using a hanging indent the following considerations should be applied; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

续行应该与其包裹元素对齐，要么使用圆括号、方括号和花括号内的隐式行连接来垂直对齐；要么使用悬挂式缩进对齐。使用悬挂式缩进时，应该考虑下面的意见。第一行不应该有参数；使用缩进以便与其他代码区分清楚。

Yes:

# Aligned with opening delimiter

# 与开始的分隔符对齐

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

# More indentation included to distinguish this from the rest.

# 更多的缩进，更加的清晰

def long\_function\_name(

var\_one, var\_two, var\_three,

var\_four):

print(var\_one)

No:

# Arguments on first line forbidden when not using vertical alignment

# 没有使用垂直对齐时，禁止把参数放在第一行

foo = long\_function\_name(var\_one, var\_two,

var\_three, var\_four)

# Further indentation required as indentation is not distinguishable

# 缩进不够，代码不清晰

def long\_function\_name(

var\_one, var\_two, var\_three,

var\_four):

print(var\_one)

Optional:

可选:

# Extra indentation is not necessary.

foo = long\_function\_name(

var\_one, var\_two,

var\_three, var\_four)

[**Tabs or Spaces? 制表符或者空格？**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id13)

Never mix tabs and spaces.

绝不要混用制表符和空格。

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. 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!

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。使用带-t选项来调用Python命令行解释器时，代码中非法混用制表符和空格会给出警告。当使用-tt选项时这些警告就会变成错误。高度建议使用这些选项。

For new projects, spaces-only are strongly recommended over tabs. Most editors have features that make this easy to do.

对新的项目，强烈建议只使用空格而不是制表符。大多数的编辑器都能轻松做到这一点。

[**Maximum Line Length 行的最大长度**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id14)

Limit all lines to a maximum of 79 characters.

限制所有行的最大长度为 79 个字符。

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 disrupts the visual structure of the code, making it more difficult to understand. Therefore, please limit all lines to a maximum of 79 characters. For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

周围还有很多设备限制每行 80个字符。而且，限制窗口显示 80 个字符可以并排显示多个窗口。有些设备上，默认折行会打乱代码结构，不易理解。因此，请限制所有行的最大长度为 79 个字符。对一大段长文本（文档字符串或注释），推荐限制每行最多72个字符。

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces. Long lines can be broken over multiple lines by wrapping expressions in parentheses. These should be used in preference to using a backslash for line continuation. Make sure to indent the continued line appropriately. The preferred place to break around a binary operator is *after* the operator, not before it. Some examples:

较长代码行折行的首选方法是在圆括号、方括号和花括号内使用Python的隐式续行方式。通过圆括号内的表达式的折行来把较长的代码行折成多行。这种方式要优先使用，优先于反斜杠。同时确保适当的续行缩进。二元运算符的首选的折行处是在运算符*之后*，而不是之前。请看一些例子：

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)

[**Blank Lines 空行**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id15)

Separate top-level function and class definitions with two blank lines.

顶层函数和类之间使用两个空行。

Method definitions inside a class are separated by a single blank line.

类的方法之间使用一个空行。

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).

（谨慎地）使用额外的空白行来分隔一组相关的函数。一堆相关的单行代码之间的空白行可以省略（例如，一组dummy implementations）。

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

在函数中使用空行来（谨慎地）表示不同的逻辑段落。

Python accepts the control-L (i.e. ^L) form feed character as whitespace; Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file. Note, some editors and web-based code viewers may not recognize control-L as a form feed and will show another glyph in its place.

Python接受control-L（即^L）换页符作为空格；许多工具把这些字符当作页面分隔符，所以你可以在文件中使用它们来分隔相关段落。请注意，一些编辑器和基于Web的代码阅读器可能无法识别control-L为换页，将在其位置显示另一个字形。

**Encodings (**[**PEP 263**](http://python.org/dev/peps/pep-0263)**) 编码**

Code in the core Python distribution should always 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](http://python.org/dev/peps/pep-3120).

Python核心发行代码应该总是使用 ASCII 或 Latin-1 编码 (又名 ISO-8859-1)。Python 3.0 和以后的版本，首选 UTF-8 而不是 Latin-1，请查看 [PEP 3120](http://python.org/dev/peps/pep-3120) 获取更多信息。（译者注：对中文项目，请首选UTF-8。）

ASCII 编码的文件不应该有coding cookie（译者注：# -\*- coding: utf-8 -\*-，# coding=utf-8）。Latin-1（或 UTF-8 ）只应该当注释或文档字符串涉及作者名需要 Latin-1 时才使用；否则，使用 \x，\u或 \U等转义字符是在字符串中包含非 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](http://python.org/dev/peps/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 (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names.

对Python 3.0 和以后的版本来说，下述政策仅对标准库作出限制（请查阅[PEP 3131](http://python.org/dev/peps/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 (a) test cases testing the non-ASCII features, and (b) names of authors. Authors whose names are not based on the latin alphabet MUST provide a latin transliteration of their names.

Open source projects with a global audience are encouraged to adopt a similar policy.

面向全球用户的开源项目，鼓励采用类似的政策。

[**Imports 导入**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id17)

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

导入通常应当使用单独的行，例如：

Yes: import os

import sys

No: import sys, os

It's okay to say this though:

但是可以这么做：

from subprocess import Popen, PIPE

* Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

导入总是位于文件的顶部，在模块注释和文档字符串之后，在模块的全局变量与常量之前。

Imports should be grouped in the following order:

导入应该按照以下的顺序分组：

* 1. standard library imports 标准库导入
  2. related third party imports 相关第三方导入
  3. local application/library specific imports 本地应用程序/库的特定导入

You should put a blank line between each group of imports.

每组导入之间使用空行隔开。

Put any relevant \_\_all\_\_ specification after the imports.

在导入之后放置任何相关的 \_\_all\_\_ 说明书。

* Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports. Even now that [PEP 328](http://python.org/dev/peps/pep-0328) is fully implemented in Python 2.5, its style of explicit relative imports is actively discouraged; absolute imports are more portable and usually more readable.

非常不推荐在包内导入中使用相对路径导入。对所有导入来说，总是使用绝对包路径导入。尽管[PEP 328](http://python.org/dev/peps/pep-0328) 在Python 2.5中已经完全实现，但是相对路径导入一直以来都是不推荐的，绝对路径导入更容易移植，而且常常更容易阅读。

* 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，常常这么写：

from myclass import MyClass

from foo.bar.yourclass import YourClass

If this spelling causes local name clashes, then spell them

如果上述写法导致内部名称冲突，那么可以这么写：

import myclass

import foo.bar.yourclass

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

分别这样使用："myclass.MyClass" 和 "foo.bar.yourclass.YourClass"。

[**Whitespace in Expressions and Statements 表达式和语句中的空格**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id18)

[**Pet Peeves 宠物的烦恼（即无伤大雅的小问题）**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id19)

Avoid extraneous whitespace in the following situations:

在下列情况下，避免多余的空格：

* Immediately inside parentheses, brackets or braces.

紧贴着圆括号、方括号和花括号。

Yes: spam(ham[1], {eggs: 2})

No: spam( ham[ 1 ], { eggs: 2 } )

* Immediately before a comma, semicolon, or colon:

紧贴在逗号，分号或冒号之前：

Yes: if x == 4: print x, y; x, y = y, x

No: if x == 4 : print x , y ; x , y = y , x

* Immediately before the open parenthesis that starts the argument list of a function call:

紧贴在函数调用的参数列表的圆括号的开括号前：

Yes: spam(1)

No: spam (1)

* Immediately before the open parenthesis that starts an indexing or slicing:

紧贴在索引或切片的方括号的开括号前：

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

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

* More than one space around an assignment (or other) operator to align it with another.

在赋值（或其他）语句的运算符周围，不要为了对齐而使用多个空格。

Yes:

x = 1

y = 2

long\_variable = 3

No:

x = 1

y = 2

long\_variable = 3

[**Other Recommendations 其他建议**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id20)

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

总是在这些二元运算符前后加一个空格：赋值（=），自增赋值（+=，-= 等等）， 比较（==，<，>，!=，<>，<=，>=，in，not in，is，is not)，布尔运算（and，or，not）。

* Use spaces around arithmetic operators:

在算术运算符前后加空格：

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.

不要在一个关键字参数或者一个缺省参数值的 = 符号前后加一个空格。

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)

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

通常不推荐使用复合语句（一行代码中有多条语句）。

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!

通常情况下，一行代码包括一个小的if/for/while块，是可以的。但是多子句绝不可以。同样，需要避免折叠类似的长代码行！

Rather not:

if foo == 'blah': do\_blah\_thing()

for x in lst: total += x

while t < 10: t = delay()

Definitely not:

绝对不可以：

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()

[**Comments 注释**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id21)

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!

不好理解的注释不如没有注释。注释要和代码保持与时俱进！

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!).

注释应该是一条完整的句子。如果注释是一个短语或句子，它的第一个字应该大写，除非它是一个小写字母开头的标识符（绝对不要改变标识符的大小写）。

If a comment is short, the period at the end can be 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，那么结尾的句号可以省略。块注释通常由一个或多个段落组成， 这些段落由完整的句子构成，每个句子都应该使用句号结尾。

You should use two spaces after a sentence-ending period.

句号结尾的句子后面应该有2个空格。

When writing English, Strunk and White apply.

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.

来自非英语国家的Python程序员：请使用英语写注释，除非你120％肯定你的代码将永远不会 被不说你的语言的人阅读。

[**Block Comments 块注释**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id22)

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).

块注释在一些（或全部）代码之前，并和代码缩进一致。每行注释均以 # 开头，然后紧跟一个空格（除非在注释内缩进）。

Paragraphs inside a block comment are separated by a line containing a single #.

块注释内的段落使用仅含 # 的单行分隔。

[**Inline Comments 行内注释**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id23)

Use inline comments sparingly.

谨慎地使用内嵌注释

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

内嵌注释是一种和语句在同一行的注释。内嵌注释至少和语句间隔2个空格。他们开始于一个 # 和一个空格。

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

如果语句显而易见，那么内嵌注释是不必要的，实际上会让开发者分心。不要这么做：

x = x + 1 # Increment x

But sometimes, this is useful:

但是有的情况下，这样却很有帮助：

x = x + 1 # Compensate for border

[**Documentation Strings 文档字符串**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id24)

Conventions for writing good documentation strings (a.k.a. "docstrings") are immortalized in [PEP 257](http://python.org/dev/peps/pep-0257).

编写良好的文档字符串（documentation strings，又名 "docstrings"）的约定永生（:(）在 [PEP 257](http://python.org/dev/peps/pep-0257)。

* 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.

所有的公告模块，函数，类和方法都需要编写文档字符串。对非公共的方法而言，文档字符串则不是必要的，但是可以使用一个注释来描述这个方法。这个注释必须位于 def 行之后。

* [PEP 257](http://python.org/dev/peps/pep-0257) describes good docstring conventions. Note that most importantly, the """ that ends a multiline docstring should be on a line by itself, and preferably preceded by a blank line, e.g.:

[PEP 257](http://python.org/dev/peps/pep-0257) 描述了一个良好的文档字符串的编写约定。注意最重要的是，""" 作为多行的文档字符串的结束，应该单独一行，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来说，结尾的 """ 在同一行。

[**Version Bookkeeping 版本注记**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id25)

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

如果在代码源文件中掺杂着Subversion，CVS或RCS注记，按照下面的规则书写。

\_\_version\_\_ = "$Revision: dc5a21b8bc47 $"

# $Source$

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

这些行在块注释之后，其他代码之前。前后各有一个空行。

[**Naming Conventions 命名约定**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id26)

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 third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

Python库的命名约定有点乱，所以我们从未达成完全一致。但是，目前有一些推荐的标准。 新的模块和包（包括第三方框架）应该用这些标准，但是拥有不同风格的固有库，还是保持内部统一更好。

[**Descriptive: Naming Styles 描述：命名风格**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id27)

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。它能够识别正在使用什么样的命名风格，而不考虑他们为什么使用。

The following naming styles are commonly distinguished:

下面是常见的优秀命名方式：

* b (single lowercase letter) （单个小写字母）
* B (single uppercase letter) （单个大写字母）
* lowercase 小写字母
* lower\_case\_with\_underscores 使用下划线分隔的小写字母
* UPPERCASE 大写字母
* UPPER\_CASE\_WITH\_UNDERSCORES 使用下划线分隔的大写字母
* CapitalizedWords (or CapWords, or CamelCase -- so named because of the bumpy look of its letters [[4]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id8)). This is also sometimes known as StudlyCaps.

CapitalizedWords （或 CapWords，或 CamelCase -- 这样命名因为其字母有起伏的外观[[4]](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id8)）。这有时也被称为 StudlyCaps。

Note: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus HTTPServerError is better than HttpServerError.

注意：当首字大写中用到缩写时，所有字符用大写。因此HTTPSeverError比HttpSeverError好。

* mixedCase (differs from CapitalizedWords by initial lowercase character!)

mixedCase（不同于首字符大写，开始是小写字母！）

* Capitalized\_Words\_With\_Underscores (ugly!)（丑陋之极！）

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. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

也有用唯一短前缀来组织相关字符的方法。在Python中用的不多。但为了完整提一下。比如，os.stat()返回一个元组， 里面的条目是传统的名字，如st\_mode, st\_size, st\_mtime 等。 (这么做是为了与 POSIX 系统的调用一致，以帮助程序员熟悉它。)

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.

X11库的所有公共函数用了前导X。在Python中，这种风格通常被视为非必须的，因为属性和方法的名字用类做前缀，方法名用模块名做前缀。

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

另外，下面用前导和结尾下划线的特殊格式是被认可的（通常和一些示例规范结合）

* \_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：（单下划线开始）弱"内部使用"指示器。例如：from M import \* 不会导入以下划线开始的对象。

* single\_trailing\_underscore\_: used by convention to avoid conflicts with Python keyword, e.g.

single\_trailing\_underscore\_：（单下划线结束）规定使用其来避免与Python关键字冲突，例如：

Tkinter.Toplevel(master, class\_='ClassName')

* \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes \_FooBar\_\_boo; see below).

\_\_double\_leading\_underscore：（双下划线开始）命名一个类的属性时，调用"名字变化"（类FooBar中，\_\_boo 变为了 \_FooBar\_\_boo; 见下文）。

* \_\_double\_leading\_and\_trailing\_underscore\_\_: "magic" objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_, \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented.

\_\_double\_leading\_and\_trailing\_underscore\_\_：（双下划线开始和结束）"magic"对象或属性存活于用户控制的命名空间。 例如\_\_init\_\_, \_\_import\_\_ 或 \_\_file\_\_。永远不要起这样的名字；只在文档中使用。

[**Prescriptive: Naming Conventions 说明：命名规范**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id28)

[**Names to Avoid 避免使用的名字**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id29)

Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.

永远不要使用 'l'（小写的L），'O'（大写的O），或者'I'（大写的I）作为单字变量名。

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use 'l', use 'L' instead.

在某些字体中，这些字很难和数字的0 和 1 区分。当打算用'l'的时候,用'L'来代替。

[**Package and Module Names 包和模块名称**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id30)

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

模块应该用简短的，全小写的名字。如果能增强可读性的话，可以使用下划线。 Python的包也要用全小写的，短名称，但是不建议用下划线。

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 older Mac or Windows versions, or DOS.

因为模块名称和文件名关联，而且某些文件系统大小写不敏感，也会截断过长的名字。所以模块选用相当简短的名字是很重要的。 在Unix下不会有这样的问题，但是在早些的Mac、Windows 或者 DOS下会有这样的问题。

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).

当用C或C++连编写一个含有Python模块提供更高层（比如，更加面向对象）接口的扩展模块时，这个C/C++模块要有一个前导下划线（例如 \_socket）。

[**Class Names 类名**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id31)

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

毫无例外，类名要用首字母大写的规则。内部类，要加上前导下划线。

[**Exception Names 异常名**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id32)

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix "Error" on your exception names (if the exception actually is an error).

异常也是类，所以这里也用类名规则。但是，你异常名应该用前缀 "Error"（如果你的异常确实是个错误的话）。

[**Global Variable Names 全局变量名**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id33)

(Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

(我们希望这些变量只在模块内部使用)这些规则和函数规则一样。

Modules that are designed for use via from M import \* should use the \_\_all\_\_ mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are "module non-public").

被设计为通过from M import \*导入的模块。应该用\_\_all\_\_机制来防止导出全局。 或者使用过去的全局变量前置下划线的规则（这是为了说明这些变量是"模块私有的"）。

[**Function Names 函数名**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id34)

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

函数名应该用小写，为了增加可读性可以用下划线分隔。

mixedCase is allowed only in contexts where that's already the prevailing style (e.g. threading.py), to retain backwards compatibility.

大小写混合仅在为了兼容原有大小写混合风格占主体的情况下使用（比如 threading.py）。

[**Function and method arguments 函数和方法参数**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id35)

Always use self for the first argument to instance methods.

始终用self作为实例方法的第一个参数。

Always use cls for the first argument to class methods.

始终用cls作为类方法的第一个参数。

If a function argument's name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class\_ is better than clss. (Perhaps better is to avoid such clashes by using a synonym.)

如果函数的参数名和保留字冲突。用结尾下划线比缩写或是滥用的组词更好。因此 class\_ 比 clss好。（也许，更好的避免冲突的方式是用同义词。）

[**Method Names and Instance Variables 方法名和实例变量**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id36)

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

使用函数命名规则：使用下划线分隔的小写字母会提高可读性。

Use one leading underscore only for non-public methods and instance variables.

只在私有方法和实例变量前用单下划线。

To avoid name clashes with subclasses, use two leading underscores to invoke Python's name mangling rules.

使用前导双下划线调用Python的名字变化规则来避免与子类中的名字冲突。

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.

Python 用类名来变化这个名字：如果类Foo有个属性叫做\_\_a,那么它不能用 Foo.\_\_a读取。 （执着的用户还是可以用Foo.\_Foo\_\_a.读取。）

Note: there is some controversy about the use of \_\_names (see below).

注意：关于\_\_names的使用有一些争论（见下文）。

[**Constants 常量**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id37)

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include MAX\_OVERFLOW and TOTAL.

常量通常在模块级别中定义，用全大写和下划线分隔的字符来编写。例如MAX\_OVERFLOW 和 TOTAL。

[**Designing for inheritance 继承设计**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id38)

Always decide whether a class's methods and instance variables (collectively: "attributes") should be public or non-public. If in doubt, choose non-public; it's easier to make it public later than to make a public attribute non-public.

总要考虑一个类的方法或实例变量（总体而言：属性）应该是公用的或者非公用的。如果不能确定的话，设计为私有的。 因为，之后将一个私有的改为公有的要比将公有的变为私有的容易。

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won't change or even be removed.

公有属性，是你希望与这个类无关的客户使用的，并通过你的委托机制来避免由于属性变更导致的向后不兼容。 私有属性，是你不希望被第三方使用的。你不能保证私有属性不会改变甚至被删除。

We don't use the term "private" here, since no attribute is really private in Python (without a generally unnecessary amount of work).

在这里我们没有使用术语"private"，因为在Python并没有真正的私有属性（为了避免大量不必要的常规工作）

Another category of attributes are those that are part of the "subclass API" (often called "protected" in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

另一种属性，是"subclass API" 的一部分（通常，在某些语言里被称作"protected"）。有一些类设计的时候是用来继承的。 用于扩展或者改变类的自身行为的某些方面。当设计这样的类的时候，一定要弄清楚哪些属性是公有的，哪些是subclass API的子集，哪些是真正只在自己的基类中使用的

With this in mind, here are the Pythonic guidelines:

贯穿着这样的思想，如下是Python的准则：

* Public attributes should have no leading underscores.

公有属性不应该用下划线开始。

* If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, 'cls' is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

如果你的公有属性和保留字冲突。那么在属性名后面加一个下划线。这要比缩写和糟糕的拼写好很多。 （然而，尽管有这样的规则，'cls'在用作变量，或者参数时，是代表'类'的最佳选择。特别是用作类方法的第一个参数）

Note 1: See the argument name recommendation above for class methods.

注意1：参看之前的类方法参数命名建议

* For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

对于单一的公有数据属性，最好是直接用其属性名，而非复杂的存取器/突变期方法。 需要记得的是，Python为今后由简单的数据属性扩充为方法操作提供了简单的途径。 因此，运用属性来实现简单数据存取语法对于方法实现的透明化处理。

Note 1: Properties only work on new-style classes.

注意1:属性只在new-style 类中起作用

Note 2: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

尽管方法行为对于缓存等的影响通常比较小，但仍要尽量避免

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

属性标记让调用者认为存取开销（相当的）小。所以，避免用属性进行开销昂贵的操作。

* If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

如果你的类打算用来继承的话，并且你的属性不希望子类继承，那么考虑用双下划线开头，不要有结尾下划线。这样会调用 Python的"名称变化术"算法，它会把类的名字错位成属性名。这样可以帮助我们避免在子类中不小心包含了相同的名字造成的冲突。

Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

注意1:要注意的是只有简单类名（simple class name）才能在名称错位中使用。 所以，如果一个子类同时用相同的类名和属性名，还是会造成名称冲突。

Note 2: Name mangling can make certain uses, such as debugging and \_\_getattr\_\_(), less convenient. However the name mangling algorithm is well documented and easy to perform manually.

"名称变化术"可用作某些用途，比如调试，\_\_getattr\_\_(),不太方便。 但是"名称变化术"有详实的文档，而且容易操作。

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

注意3:并不是所有人喜欢"名称变化术"。尝试平衡需求，避免由于之前潜在的调用造成的意外的名称冲突。

[**Programming Recommendations  程序设计建议**](http://www.cnblogs.com/ajianbeyourself/p/4377933.html#id39)

* Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).

代码应该用不损害其他Python实现的方式去编写（PyPy, Jython, IronPython, Cython, Psyco 等）

For example, do not rely on CPython's efficient implementation of in-place string concatenation for statements in the form a += b or a = a + b. Those statements run more slowly in Jython. In performance sensitive parts of the library, the ''.join() form should be used instead. This will ensure that concatenation occurs in linear time across various implementations.

例如，不要依赖于CPython的高效内置字符连接语句a += b or a = a + b.这些语句在Jython中运行较慢。在性能敏感的库中，应该用''.join() 来取代。这样可以保证在不同的实现中，字符链接花费的时间都呈线性。

* Comparisons to singletons like None should always be done with is or is not, never the equality operators.

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 have a type (such as a container) that could be false in a boolean context!

与诸如None这样的字符比较时，要使用is or is not，永远不要用等于操作。

同样地，在测试一个变量或参数默认值为None被设置为其他值时（比如if x表示if x is not None时），要注意。 这个值应该有一个能在布尔逻辑的上下文中为false的类型（比如容器）。

* When implementing ordering operations with rich comparisons, it is best to implement all six operations (\_\_eq\_\_, \_\_ne\_\_, \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_) rather than relying on other code to only exercise a particular comparison.

当用复杂比较实现排序操作时，最好去实现全部六个操作(\_\_eq\_\_, \_\_ne\_\_, \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_)，而不是依靠其他的代码去实现一些怪异的比较。

To minimize the effort involved, the functools.total\_ordering() decorator provides a tool to generate missing comparison methods.

为了最大程度的减少实现这一过程的开销。functools.total\_ordering()提供了生成缺少的比较操作的方法。

[PEP 207](http://python.org/dev/peps/pep-0207) indicates that reflexivity rules *are* assumed by Python. Thus, the interpreter may swap y > x with x < y, y >= x with x <= y, and may swap the arguments of x == y and x != y. The sort() and min() operations are guaranteed to use the < operator and the max() function uses the > operator. However, it is best to implement all six operations so that confusion doesn't arise in other contexts.

[PEP 207](http://python.org/dev/peps/pep-0207)指出Python实现了反射。因此，解释器会将y > x 转换为 x < y, y >= x转换为 x <= y，也会转换x == y 和 x != y的参数.The sort() 和 min()被确保使用<操作符，max()使用>操作符。 但是，最好实现全部六个操作，以避免在其他地方造成的冲突。

* Use class-based exceptions.

使用基于对象的异常

String exceptions in new code are forbidden, because this language feature is being removed in Python 2.6.

字符异常在新的代码中被禁止了，因为这一语言特性在2.6版本中被移除了。

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.:

模块或者包应该定义自己的异常基类，这个类应该继承自内置的Exception类。总要包含类文档语句，例如：

class MessageError(Exception):

"""Base class for errors in the email package."""

Class naming conventions apply here, although you should add the suffix "Error" to your exception classes, if the exception is an error. Non-error exceptions need no special suffix.

在这里用到类命名规范。尽管当你的“异常”是一个“错误”（error）时，你应该在自己的异常类加上后缀"Error"。 非错误类（Non-error）异常，不需要特别的后缀。

* When raising an exception, use raise ValueError('message') instead of the older form raise ValueError, 'message'.

当抛出一个异常的时候，使用raise ValueError('message')代替旧的raise ValueError, 'message'格式。

The paren-using form is preferred because when the exception arguments are long or include string formatting, you don't need to use line continuation characters thanks to the containing parentheses. The older form will be removed in Python 3.

推荐使用这种括弧的格式，当异常的参数很长或者是格式化字符串的时候，由于括号的关系，我们不需要使用连字符。 旧的格式在Python 3 中被移除。

* When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause.

当捕获一个异常的时候，要用详细异常声明代替光秃秃的except: 语句.

For example, use:

try:

import platform\_specific\_module

except ImportError:

platform\_specific\_module = None

A bare except: clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use except Exception: (bare except is equivalent to except BaseException:).

一个空的except:语句将会捕获 SystemExit 和 KeyboardInterrrupt 异常。这会使得很难用Control-C来中断一个程序，并且还会隐藏其他的问题。 如果你想捕获一个程序中的所有异常，使用except Exception: （bare 和except BaseException:是等价的）。

A good rule of thumb is to limit use of bare 'except' clauses to two cases:

经验告诉我们在两个情况下避免使用空‘except’

* 1. If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.

如果异常处理程序要打印或者记录回溯。至少使用者能意识到错误已经发生了。

* 1. If the code needs to do some cleanup work, but then lets the exception propagate upwards with raise. try...finally can be a better way to handle this case.

如果代码要做一些收尾工作，但是随后要用 raise向上抛出异常。try...finally可以更好地处理这个问题。

* Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs.

另外，对于所有的try/except语句，限制try语句来减少必要代码的数量。 再者，可以避免掩盖问题。

Yes:

try:

value = collection[key]

except KeyError:

return key\_not\_found(key)

else:

return handle\_value(value)

No:

try:

# Too broad!

return handle\_value(collection[key])

except KeyError:

# Will also catch KeyError raised by handle\_value()

return key\_not\_found(key)

* Use string methods instead of the string module.

使用String方法而非String模块

String methods are always much faster and share the same API with unicode strings. Override this rule if backward compatibility with Pythons older than 2.0 is required.

String方法更加快捷并且使用和unicode字符一样的API。如果需要向后兼容2.0之前的版本的话，可以不考虑这条规则。

* Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

使用''.startswith() 和 ''.endswith()而非字符切片去检测前缀或后缀。

startswith() and endswith() are cleaner and less error prone. For example:

startswith() and endswith() 更加绿色，错误更少。例如：

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

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

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

例外的就是如果你的代码必须在Python 1.5.2运行（但是，假设它不是）。

* Object type comparisons should always use isinstance() instead of comparing types directly.

对象类型比较总要用 isinstance() 而非直接比较

Yes: if isinstance(obj, int):

No: if type(obj) is type(1):

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:

当检测一个对象是否是字符串是，记住，它可能是unicode字符。在Python 2.3， str和unicode有同样的基类——basestring, 那么你可以这样做：

if isinstance(obj, basestring):

* For sequences, (strings, lists, tuples), use the fact that empty sequences are false.

对于序列，(strings, lists, tuples)，利用空序列为false这一点。

Yes: if not seq:

if seq:

No: if len(seq)

if not len(seq)

* 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.

书写字面值时不要依靠后面的空格。这些后面的空格，视觉上难以区分，而且很多编辑器（或者，眼前的，reindent.py）会去掉他们。

* Don't compare boolean values to True or False using ==.

别用‘==’进行布尔值和 True 或者 False 的比较

Yes: if greeting:

No: if greeting == True:

Worse: if greeting is True: