Python 编码规范(Google)

分类 **编程技术**

Python 风格规范(Google)
本项目并非 Google 官方项目, 而是由国内程序员凭热情创建和维护。
如果你关注的是 Google 官方英文版, 请移步 Google Style Guide
以下代码中 **Yes** 表示推荐，**No** 表示不推荐。

分号

不要在行尾加分号, 也不要用分号将两条命令放在同一行。

行长度

每行不超过80个字符
以下情况除外:

1. 长的导入模块语句
2. 注释里的URL

不要使用反斜杠连接行。
Python会将 圆括号, 中括号和花括号中的行隐式的连接起来 , 你可以利用这个特点. 如果需要, 你可以在表达式外围增加一对额外的圆括号。

```
推荐: foo_bar(self, width, height, color='black', design=None, x='foo',
              emphasis=None, highlight=0)

    if (width == 0 and height == 0 and
        color == 'red' and emphasis == 'strong'):
```

如果一个文本字符串在一行放不下, 可以使用圆括号来实现隐式行连接:

```
x = ('这是一个非常长非常长非常长非常长 '
     '非常长非常长非常长非常长非常长非常长的字符串')
```

在注释中，如果必要，将长的URL放在一行上。

```
Yes:  # See details at
      # http://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_name_extension_full_specifica
tion.html
```

```
No:  # See details at
     # http://www.example.com/us/developer/documentation/api/content/\
     # v2.0/csv_file_name_extension_full_specification.html
```

注意上面例子中的元素缩进; 你可以在本文的 :ref:`缩进 <indentation>`部分找到解释.

括号

宁缺毋滥的使用括号
除非是用于实现行连接, 否则不要在返回语句或条件语句中使用括号. 不过在元组两边使用括号是可以的.

```
Yes: if foo:
        bar()
     while x:
         x = bar()
     if x and y:
         bar()
     if not x:
         bar()
```

教程列表

ADO 教程 Ajax 教程 Android 教程
Angular2 教程 AngularJS 教程 AppML 教程
ASP 教程 ASP.NET 教程 Bootstrap 教程
Bootstrap4 教程 C 教程 C# 教程
C++ 教程 CSS 参考手册 CSS 教程
CSS3 教程 Django 教程 Docker 教程
DTD 教程 ECharts 教程 Eclipse 教程
Firebug 教程 Font Awesome 图 Foundation 教程
Git 教程 Go 语言教程 Google 地图 API
Highcharts 教程 HTML DOM 教程 HTML 参考手册
HTML 字符集 HTML 教程 HTTP 教程
ionic 教程 iOS 教程 Java 教程
JavaScript 参考手 Javascript 教程 jQuery EasyUI 教
jQuery Mobile 教 jQuery UI 教程 jQuery 教程
JSON 教程 JSP 教程 Kotlin 教程
Linux 教程 Lua 教程 Markdown 教程
Maven 教程 Memcached 教程 MongoDB 教程
MySQL 教程 Node.js 教程 NumPy 教程
Perl 教程 PHP 教程 PostgreSQL 教程
Python 3 教程 Python 基础教程 R 教程
RDF 教程 React 教程 Redis 教程
RSS 教程 Ruby 教程 Rust 教程
Sass 教程 Scala 教程 Servlet 教程
SOAP 教程 SQL 教程 SQLite 教程
SVG 教程 SVN 教程 Swift 教程
TCP/IP 教程 TypeScript 教程 VBScript 教程
Vue.js 教程 W3C 教程 Web Service 教程
WSDL 教程 XLink 教程 XML DOM 教程
XML Schema 教 XML 教程 XPath 教程
XQuery 教程 XSLFO 教程 XSLT 教程
正则表达式 测验 浏览器
网站品质 网站建设指南 网站服务器教程
设计模式

```
    return foo
    for (x, y) in dict.items(): ...
```

```
No:  if (x):
        bar()
    if not(x):
        bar()
    return (foo)
```

缩进

用4个空格来缩进代码

绝对不要用tab, 也不要tab和空格混用. 对于行连接的情况, 你应该要么垂直对齐换行的元素(见 :ref:`行长度 <line_length>` 部分的示例), 或者使用4空格的悬挂式缩进(这时第一行不应该有参数):

```
Yes:   # 与起始变量对齐
       foo = long_function_name(var_one, var_two,
                                var_three, var_four)

       # 字典中与起始值对齐
       foo = {
           long_dictionary_key: value1 +
                                value2,
           ...
       }

       # 4 个空格缩进，第一行不需要
       foo = long_function_name(
           var_one, var_two, var_three,
           var_four)

       # 字典中 4 个空格缩进
       foo = {
           long_dictionary_key:
               long_dictionary_value,
           ...
       }
```

```
No:    # 第一行有空格是禁止的
      foo = long_function_name(var_one, var_two,
          var_three, var_four)

      # 2 个空格是禁止的
      foo = long_function_name(
        var_one, var_two, var_three,
        var_four)

      # 字典中没有处理缩进
      foo = {
          long_dictionary_key:
              long_dictionary_value,
              ...
      }
```

空行

顶级定义之间空两行, 方法定义之间空一行

顶级定义之间空两行, 比如函数或者类定义. 方法定义, 类定义与第一个方法之间, 都应该空一行. 函数或方法中, 某些地方要是你觉得合适, 就空一行.

空格

按照标准的排版规范来使用标点两边的空格

括号内不要有空格.

按照标准的排版规范来使用标点两边的空格

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

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

不要在逗号, 分号, 冒号前面加空格, 但应该在它们后面加(除了在行尾).

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

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

参数列表, 索引或切片的左括号前不应加空格.

```
Yes: spam(1)
```

```
no: spam (1)
```

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

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

在二元操作符两边都加上一个空格, 比如赋值(=), 比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not), 布尔(and, or, not). 至于算术操作符两边的空格该如何使用, 需要你自己好好判断. 不过两侧务必要保持一致.

```
Yes: x == 1
```

```
No:  x<1
```

当'='用于指示关键字参数或默认参数值时, 不要在其两侧使用空格.

```
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)
```

不要用空格来垂直对齐多行间的标记, 因为这会成为维护的负担(适用于:, #, =等):

```
Yes:
    foo = 1000  # 注释
    long_name = 2  # 注释不需要对齐

    dictionary = {
        "foo": 1,
        "long_name": 2,
        }
```

```
No:
    foo       = 1000  # 注释
    long_name = 2     # 注释不需要对齐

    dictionary = {
        "foo"      : 1,
        "long_name": 2,
        }
```

Shebang

大部分.py文件不必以#!作为文件的开始. 根据 PEP-394 , 程序的main文件应该以 #!/usr/bin/python2或者 #!/usr/bin/python3开始. (译者注: 在计算机科学中, Shebang (也称为Hashbang)是一个由井号和叹号构成的字符串行(#!), 其出现在文本文件的第一行的前两个字符. 在文件中存在Shebang的情况下, 类Unix操作系统的程序载入器会分析Shebang后的内容, 将这些内容作为解释器指令, 并调用该指令, 并将载有Shebang的文件路径作为该解释器的参数. 例如, 以指令#!/bin/sh开头的文件在执行时会实际调用/bin/sh程序.)

#!先用于帮助内核找到Python解释器, 但是在导入模块时, 将会被忽略. 因此只有被直接执行的文件中才有必要加入#!.

注释

确保对模块, 函数, 方法和行内注释使用正确的风格

文档字符串

Python有一种独一无二的的注释方式: 使用文档字符串. 文档字符串是包, 模块, 类或函数里的第一个语句. 这些字符串可以通过对象的__doc__成员被自动提取, 并且被pydoc所用. (你可以在你的模块上运行pydoc试一把, 看看它长什么样). 我们对文档字符串的惯例是使用三重双引号"""(PEP-257). 一个文档字符串应该这样组织: 首先是一行以句号, 问号或惊叹号结尾的概述(或者该文档字符串单纯只有一行). 接着是一个空行. 接着是文档字符串剩下的部分, 它应该与文档字符串的第一行的第一个引号对齐. 下面有更多文档字符串的格式化规范.

模块

每个文件应该包含一个许可样板. 根据项目使用的许可(例如, Apache 2.0, BSD, LGPL, GPL), 选择合适的样板.

函数和方法

下文所指的函数,包括函数, 方法, 以及生成器.

一个函数必须要有文档字符串, 除非它满足以下条件:

1. 外部不可见
2. 非常短小
3. 简单明了

文档字符串应该包含函数做什么, 以及输入和输出的详细描述. 通常, 不应该描述"怎么做", 除非是一些复杂的算法. 文档字符串应该提供足够的信息, 当别人编写代码调用该函数时, 他不需要看一行代码, 只要看文档字符串就可以了. 对于复杂的代码, 在代码旁边加注释会比使用文档字符串更有意义.

关于函数的几个方面应该在特定的小节中进行描述记录, 这几个方面如下文所述. 每节应该以一个标题行开始. 标题行以冒号结尾. 除标题行外, 节的其他内容应被缩进2个空格.

Args:

列出每个参数的名字, 并在名字后使用一个冒号和一个空格, 分隔对该参数的描述.如果描述太长超过了单行80字符,使用2或者4个空格的悬挂缩进(与文件其他部分保持一致). 描述应该包括所需的类型和含义. 如果一个函数接受*foo(可变长度参数列表)或者**bar (任意关键字参数), 应该详细列出*foo和**bar.

Returns: (或者 Yields: 用于生成器)

描述返回值的类型和语义. 如果函数返回None, 这一部分可以省略.

Raises:

列出与接口有关的所有异常.

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
    pass
```

类

类应该在其定义下有一个用于描述该类的文档字符串. 如果你的类有公共属性(Attributes), 那么文档中应该有一个属性(Attributes)段. 并且应该遵守和函数参数相同的格式.

```
class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

块注释和行注释

最需要写注释的是代码中那些技巧性的部分. 如果你在下次 代码审查 的时候必须解释一下, 那么你应该现在就给它写注释. 对于复杂的操作, 应该在其操作开始前写上若干行注释. 对于不是一目了然的代码, 应在其行尾添加注释.

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:        # true iff i is a power of 2
```

为了提高可读性, 注释应该至少离开代码2个空格.

另一方面, 绝不要描述代码. 假设阅读代码的人比你更懂Python, 他只是不知道你的代码要做什么.

```
# BAD COMMENT: Now go through the b array and make sure whenever i occurs
# the next element is i+1
```

类

如果一个类不继承自其它类, 就显式的从object继承. 嵌套类也一样.

```
Yes: class SampleClass(object):
        pass


    class OuterClass(object):

        class InnerClass(object):
            pass


    class ChildClass(ParentClass):
        """Explicitly inherits from another class already."""
```

```
No: class SampleClass:
       pass


   class OuterClass:

       class InnerClass:
           pass
```

继承自 `object` 是为了使属性(properties)正常工作, 并且这样可以保护你的代码, 使其不受Python 3000的一个特殊的潜在不兼容性影响. 这样做也定义了一些特殊的方法, 这些方法实现了对象的默认语义, 包括 `__new__`, `__init__`, `__delattr__`, `__getattribute__`, `__setattr__`, `__hash__`, `__repr__`, and `__str__`.

字符串

```
Yes: x = a + b
     x = '%s, %s!' % (imperative, expletive)
     x = '{}, {}!'.format(imperative, expletive)
     x = 'name: %s; score: %d' % (name, n)
     x = 'name: {}; score: {}'.format(name, n)
```

```
No: x = '%s%s' % (a, b)  # use + in this case
    x = '{}{}'.format(a, b)  # use + in this case
    x = imperative + ', ' + expletive + '!'
    x = 'name: ' + name + '; score: ' + str(n)
```

避免在循环中用+和+=操作符来累加字符串. 由于字符串是不可变的, 这样做会创建不必要的临时对象, 并且导致二次方而不是线性的运行时间. 作为替代方案, 你可以将每个子串加入列表, 然后在循环结束后用 `.join` 连接列表. (也可以将每个子串写入一个 `cStringIO.StringIO` 缓存中.)

```
Yes: items = ['<table>']
     for last_name, first_name in employee_list:
         items.append('<tr><td>%s, %s</td></tr>' % (last_name, first_name))
     items.append('</table>')
     employee_table = ''.join(items)
```

```
No: employee_table = '<table>'
    for last_name, first_name in employee_list:
        employee_table += '<tr><td>%s, %s</td></tr>' % (last_name, first_name)
    employee_table += '</table>'
```

在同一个文件中, 保持使用字符串引号的一致性. 使用单引号'或者双引号"之一用以引用字符串, 并在同一文件中沿用. 在字符串内可以使用另外一种引号, 以避免在字符串中使用. PyLint已经加入了这一检查.

```
Yes:
     Python('Why are you hiding your eyes?')
     Gollum("I'm scared of lint errors.")
     Narrator('"Good!" thought a happy Python reviewer.')
```

```
No:
     Python("Why are you hiding your eyes?")
     Gollum('The lint. It burns. It burns us.')
     Gollum("Always the great lint. Watching. Watching.")
```

为多行字符串使用三重双引号"""而非三重单引号'''. 当且仅当项目中使用单引号'来引用字符串时, 才可能会使用三重'''为非文档字符串的多行字符串来标识引用. 文档字符串必须使用三重双引号"""". 不过要注意, 通常用隐式行连接更清晰, 因为多行字符串与程序其他部分的缩进方式不一致.

```
Yes:
    print ("This is much nicer.\n"
           "Do it this way.\n")
```

```
No:
      print """This is pretty ugly.
  Don't do this.
  """
```

文件和sockets

在文件和sockets结束时, 显式的关闭它.

除文件外, sockets或其他类似文件的对象在没有必要的情况下打开, 会有许多副作用, 例如:

1. 它们可能会消耗有限的系统资源, 如文件描述符. 如果这些资源在使用后没有及时归还系统, 那么用于处理这些对象的代码会将资源消耗殆尽.
2. 持有文件将会阻止对于文件的其他诸如移动、删除之类的操作.
3. 仅仅是从逻辑上关闭文件和sockets, 那么它们仍然可能会被其共享的程序在无意中进行读或者写操作. 只有当它们真正被关闭后, 对于它们尝试进行读或者写操作将会跑出异常, 并使得问题快速显现出来.

而且, 幻想当文件对象析构时, 文件和sockets会自动关闭, 试图将文件对象的生命周期和文件的状态绑定在一起的想法, 都是不现实的. 因为有如下原因:

1. 没有任何方法可以确保运行环境会真正的执行文件的析构. 不同的Python实现采用不同的内存管理技术, 比如延时垃圾处理机制. 延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长.

2. 对于文件意外的引用,会导致对于文件的持有时间超出预期(比如对于异常的跟踪, 包含有全局变量等).

推荐使用 "with"语句 以管理文件:

```
with open("hello.txt") as hello_file:
    for line in hello_file:
        print line
```

对于不支持使用"with"语句的类似文件的对象,使用 contextlib.closing():

```
import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as front_page:
    for line in front_page:
        print line
```

Legacy AppEngine 中Python 2.5的代码如使用"with"语句, 需要添加 "from __future__ import with_statement".

TODO注释

为临时代码使用TODO注释, 它是一种短期解决方案. 不算完美, 但够好了.
TODO注释应该在所有开头处包含"TODO"字符串, 紧跟着是用括号括起来的你的名字, email地址或其它标识符. 然后是一个可选的冒号. 接着必须有一行注释, 解释要做什么. 主要目的是为了有一个统一的TODO格式, 这样添加注释的人就可以搜索到(并可以按需提供更多细节). 写了TODO注释并不保证写的人会亲自解决问题. 当你写了一个TODO, 请注上你的名字.

```
# TODO(kl@gmail.com): Use a "*" here for string repetition.
# TODO(Zeke) Change this to use relations.
```

如果你的TODO是"将来做某事"的形式, 那么请确保你包含了一个指定的日期("2009年11月解决")或者一个特定的事件("等到所有的客户都可以处理XML请求就移除这些代码").

导入格式

每个导入应该独占一行

```
Yes: import os
     import sys
```

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

导入总应该放在文件顶部, 位于模块注释和文档字符串之后, 模块全局变量和常量之前. 导入应该按照从最通用到最不通用的顺序分组:

1. 标准库导入
2. 第三方库导入
3. 应用程序指定导入

每种分组中, 应该根据每个模块的完整包路径按字典序排序, 忽略大小写.

```
import foo
from foo import bar
from foo.bar import baz
from foo.bar import Quux
from Foob import ar
```

语句

通常每个语句应该独占一行
不过, 如果测试结果与测试语句在一行放得下, 你也可以将它们放在同一行. 如果是if语句, 只有在没有else时才能这样做. 特别地, 绝不要对 `try/except` 这样做, 因为try和except不能放在同一行.

```
Yes:

  if foo: bar(foo)
```

```
No:
```

```
if foo: bar(foo)
else:   baz(foo)

try:               bar(foo)
except ValueError: baz(foo)

try:
    bar(foo)
except ValueError: baz(foo)
```

访问控制

在Python中, 对于琐碎又不太重要的访问函数, 你应该直接使用公有变量来取代它们, 这样可以避免额外的函数调用开销. 当添加更多功能时, 你可以用属性(property)来保持语法的一致性.

(译者注: 重视封装的面向对象程序员看到这个可能会很反感, 因为他们一直被教育: 所有成员变量都必须是私有的! 其实, 那真的是有点麻烦啊. 试着去接受Pythonic哲学吧)

另一方面, 如果访问更复杂, 或者变量的访问开销很显著, 那么你应该使用像 `get_foo()` 和 `set_foo()` 这样的函数调用. 如果之前的代码行为允许通过属性(property)访问 , 那么就不要将新的访问函数与属性绑定. 这样, 任何试图通过老方法访问变量的代码就没法运行, 使用者也就会意识到复杂性发生了变化.

命名

module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_VAR_NAME, instance_var_name, function_parameter_name, local_var_name.

应该避免的名称

1. 单字符名称, 除了计数器和迭代器.
2. 包/模块名中的连字符(-)
3. 双下划线开头并结尾的名称(Python保留, 例如__init__)

命名约定

1. 所谓"内部(Internal)"表示仅模块内可用, 或者, 在类内是保护或私有的.
2. 用单下划线(_)开头表示模块变量或函数是protected的(使用import * from时不会包含).
3. 用双下划线(__)开头的实例变量或方法表示类内私有.
4. 将相关的类和顶级函数放在同一个模块里. 不像Java, 没必要限制一个类一个模块.
5. 对类名使用大写字母开头的单词(如CapWords, 即Pascal风格), 但是模块名应该用小写加下划线的方式(如lower_with_under.py). 尽管已经有很多现存的模块使用类似于CapWords.py这样的命名, 但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致, 这会让人困扰.

Python之父Guido推荐的规范

Type	Public	Internal
Modules	lower_with_under	_lower_with_under
Packages	lower_with_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected) or __lower_with_under (private)
Method Names	lower_with_under()	_lower_with_under() (protected) or __lower_with_under() (private)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

Main

即使是一个打算被用作脚本的文件, 也应该是可导入的. 并且简单的导入不应该导致这个脚本的主功能(main functionality)被执行, 这是一种副作用. 主功能应该放在一个main()函数中.

在Python中, pydoc以及单元测试要求模块必须是可导入的. 你的代码应该在执行主程序前总是检查 `if __name__ == '__main__'`, 这样当模块被导入时主程序就不会被执行.

```
def main():
      ...

if __name__ == '__main__':
    main()
```

所有的顶级代码在模块导入时都会被执行. 要小心不要去调用函数, 创建对象, 或者执行那些不应该在使用pydoc时执行的操作.

← 史上最全Vim快捷键键位图 — 入门到进阶 史上最全Vim快捷键键位图（入门到进阶） →