Team-Xmbot-Service-Robot Python风格指南

0. 前言

0.1 版本

- 1.0版本(2016.5.1)：缪宇飚(myyerrol)创建团队 Python 代码开发风格指南。本文档参考了 ROS 和 Google 的 Python 风格指南，并根据实际的需求，对内容进行了适当的精简和改进。新队员应该认真学习本指南，掌握 Python 基本的开发风格。如果有细节不统一的地方或者对本文档某处不是很认同，请在组内讨论统一之后，修改本指南。因为文档排版使用的是 Markdown 纯文本标记语言，也请后来者遵循本文档的开发方式，使用 Markdown 来修改、添加内容。

0.2 背景

Python 是 ROS 项目开发所使用的脚本语言。其在很多有关编写机器人策略的代码以及简单机械臂控制中应用的非常多（至少在组内是这个样子的）。尽管在语法上 Python 相较于 C++ 简单很多，但是很多惨痛的经历告诫我们，建立一套统一的 Python 代码风格是很有必要的。

Team-Xmbot-Service-Robot（晓萌家庭服务机器人团队）作为开源项目，需要团队队员贡献代码，但是如果队员之间的代码编程风格不一致，便会给团队其他模块负责人造成不小的困扰。我们认为整洁、一致的代码风格可以使代码更加可读、更加可调试以及更加可维护。因此，我们应该编写优雅的代码使其不仅能够在现在发挥作用，而且在将来的若干年之后其依旧能够存在、可以被复用、或者是能够被未来的新队员改进。

1. Python 语言规范

1.1 导入

> **Tip**
>
> 仅对包和模块使用导入。

定义：
模块间共享代码的重用机制。

优点：
命名空间管理约定十分简单。每个标识符的源都用一种一致的方式指示。x.Obj 表示 Obj 对象定义在模块 x 中。

缺点：
模块名仍可能冲突。有些模块名太长，不太方便。

结论：
使用 `import x` 来导入包和模块。
使用 `from x import y`，其中 x 是包前缀，y 是不带前缀的模块名。
使用 `from x import y as z`，如果两个要导入的模块都叫做 z 或者 y 太长了。

例如，模块 `sound.effects.echo` 可以用如下方式导入：

```
from sound.effects import echo
...
echo.EchoFilter(input, output, delay=0.7, atten=4)
```

导入时不要使用相对名称。即使模块在同一个包中，也要使用完整包名。这能帮助你避免无意间导入一个包两次。

1.2 包

> **Tip**
>
> 使用模块的全路径名来导入每个模块。

优点：
避免模块名冲突。查找包更容易。

缺点：
部署代码变难，因为你必须复制包层次。

结论：
所有的新代码都应该用完整包名来导入每个模块。

应该像下面这样导入：

```
# Reference in code with complete name.
import sound.effects.echo

# Reference in code with just module name (preferred).
from sound.effects import echo
```

1.3 异常

> **Tip**
>
> 允许使用异常，但必须小心。

定义：
异常是一种跳出代码块的正常控制流来处理错误或者其它异常条件的方式。

优点：
正常操作代码的控制流不会和错误处理代码混在一起。当某种条件发生时，它也允许控制流跳过多个框架。例如，一步跳出 N 个嵌套的函数，而不必继续执行错误的代码。

缺点：
可能会导致让人困惑的控制流。调用库时容易错过错误情况。

结论：
异常必须遵守特定条件：

- 像这样触发异常：`raise MyException("Error message")` 或者 `raise MyException`。不要使用两个参数的形式（`raise MyException, "Error message"`）或者过时的字符串异常（`raise "Error message"`）。
- 模块或包应该定义自己的特定域的异常基类，这个基类应该从内建的 `Exception` 类继承。模块的异常基类应该叫做 “Error”。

```
class Error(Exception):
    pass
```

- 永远不要使用 `except:` 语句来捕获所有异常，也不要捕获 `Exception` 或者

`StandardError`，除非你打算重新触发该异常，或者你已经在当前线程的最外层（记得还是要打印一条错误消息）。 在异常这方面，Python 非常宽容，`except:` 真的会捕获包括 Python 语法错误在内的任何错误。使用 `except:` 很容易隐藏真正的 Bug。

- 尽量减少 `try/except` 块中的代码量。`try` 块的体积越大，期望之外的异常就越容易被触发。这种情况下，`try/except` 块将隐藏真正的错误。
- 使用 `finally` 子句来执行那些无论 `try` 块中有没有异常都应该被执行的代码。这对于清理资源常常很有用，例如关闭文件。
- 当捕获异常时，使用 `as` 而不要用逗号。例如：

```
try:
    raise Error
except Error as error:
    pass
```

1.4 全局变量

> **Tip**
>
> 避免全局变量。

定义：
定义在模块级的变量。

优点：
偶尔有用。

缺点：
导入时可能改变模块行为，因为导入模块时会对模块级变量赋值。

结论：
避免使用全局变量，用类变量来代替。但也有一些例外：

- 脚本的默认选项。
- 模块级常量。例如：PI = 3.14159。常量应该全大写，用下划线连接。
- 有时候用全局变量来缓存值或者作为函数返回值很有用。
- 如果需要，全局变量应该仅在模块内部可用，并通过模块级的公共函数来访问。

1.5 列表推导

> **Tip**
>
> 可以在简单情况下使用。

定义：
列表推导与生成器表达式提供了一种简洁高效的方式来创建列表和迭代器，而不必借助 `map()`、`filter()`、或者 `lambda`。

优点：
简单的列表推导可以比其它的列表创建方法更加清晰简单。生成器表达式可以十分高效，因为它们避免了创建整个列表。

缺点：
复杂的列表推导或者生成器表达式可能难以阅读。

结论：
适用于简单情况。每个部分应该单独置于一行：映射表达式、for 语句、过滤器表达式。禁止多重 for 语句或过滤器表达式。复杂情况下还是使用循环。

```
Yes:
  result = []
  for x in range(10):
      for y in range(5):
          if x * y > 10:
              result.append((x, y))

  for x in xrange(5):
      for y in xrange(5):
          if x != y:
              for z in xrange(5):
                  if y != z:
                      yield (x, y, z)

  return ((x, complicated_transform(x))
          for x in long_generator_function(parameter)
          if x is not None)

  squares = [x * x for x in range(10)]

  eat(jelly_bean for jelly_bean in jelly_beans
      if jelly_bean.color == 'black')
```

```
No:
  result = [(x, y) for x in range(10) for y in range(5) if x * y > 10]

  return ((x, y, z)
          for x in xrange(5)
          for y in xrange(5)
          if x != y
          for z in xrange(5)
          if y != z)
```

1.6 默认迭代器和操作符

> **Tip**
>
> 如果类型支持，就使用默认迭代器和操作符。比如列表，字典及文件等。

定义：
容器类型，像字典和列表，定义了默认的迭代器和关系测试操作符。

优点：
默认操作符和迭代器简单高效，它们直接表达了操作，没有额外的方法调用。使用默认操作符的函数是通用的。它可以用于支持该操作的任何类型。

缺点：
你没法通过阅读方法名来区分对象的类型（例如 `has_key()` 意味着字典）。 不过这也是优点。

结论：
如果类型支持，就使用默认迭代器和操作符，例如列表，字典和文件。内建类型也定义了迭代器方法。优先考虑这些方法，而不是那些返回列表的方法。当然，这样遍历容器时，你将不能修改容器。

```
Yes:  for key in adict: ...
      if key not in adict: ...
      if obj in alist: ...
      for line in afile: ...
      for k, v in dict.iteritems(): ...
```

```
No:   for key in adict.keys(): ...
      if not adict.has_key(key): ...
      for line in afile.readlines(): ...
```

1.7 默认参数值

> **Tip**
>
> 适用于大部分情况。

定义：
你可以在函数参数列表的最后指定变量的值，例如：`def foo(a, b = 0):` 。如果调用 `foo` 时只带一个参数，则 `b` 被设为 0。如果带两个参数，则 `b` 的值等于第二个参数。

优点：
你经常会碰到一些使用大量默认值的函数，但偶尔（比较少见）你想要覆盖这些默认值。默认参数值提供了一种简单的方法来完成这件事，你不需要为这些罕见的例外定义大量函数。同时，Python 也不支持重载方法和函数，默认参数是一种"仿造"重载行为的简单方式。

缺点：
默认参数只在模块加载时求值一次。如果参数是列表或字典之类的可变类型，这可能会导致问题。如果函数修改了对象（例如向列表追加项），默认值就被修改了。

结论：
鼓励使用，不过有如下注意事项：

不要在函数或方法定义中使用可变对象作为默认值。

```
Yes: def foo(a, b=None):
         if b is None:
             b = []
```

```
No:  def foo(a, b=[]):
         ...
No:  def foo(a, b=time.time()):  # The time the module was loaded?
         ...
No:  def foo(a, b=FLAGS.my_thing):  # sys.argv has not yet been parse
d...
         ...
```

1.8 True/False 的求值

> **Tip**
>
> 尽可能使用隐式 `false`。

定义：
Python 在布尔上下文中会将某些值求值为 `false`。按简单的直觉来讲，就是所有的”空“值都被认为是 `false`。因此 `0`、`None`、`[]`、`{}`、`""` 都被认为是 `false`。

优点：
使用 Python 布尔值的条件语句更易读也更不易犯错。大部分情况下，也更快。

缺点：
对 C/C++ 开发人员来说，可能看起来有点怪。

结论：
尽可能使用隐式的 `false`，例如： 使用 `if foo:` 而不是 `if foo != []:`。 不过还是有一些注意事项需要你铭记在心：

- 永远不要用 `==` 或者 `!=` 来比较单件，比如 `None`。使用 `is` 或者 `is not`。
- 注意：当你写下 `if x:` 时，你其实表示的是 `if x is not None`。例如：当你要测试一个默认值是 `None` 的变量或参数是否被设为其它值。这个值在布尔语义下可能是 `false`！
- 永远不要用 `==` 将一个布尔量与 `false` 相比较。使用 `if not x:` 代替。如果你需要区分 `false` 和 `None`，你应该用像 `if not x and x is not None:` 这样的语句。
- 对于序列（字符串、列表、元组），要注意空序列是 `false`。因此 `if not seq:` 或者 `if seq:` 比 `if len(seq):` 或 `if not len(seq):` 要更好。
- 处理整数时，使用隐式 `false` 可能会得不偿失（即不小心将 `None` 当做 0 来处理）。你可以将一个已知是整型（且不是 `len()` 的返回结果）的值与 0 比较。

```
Yes: if not users:
         print 'no users'

     if foo == 0:
         self.handle_zero()

     if i % 10 == 0:
         self.handle_mulTiple_of_ten()
```

```
No:  if len(users) == 0:
         print 'no users'

     if foo is not None and not foo:
         self.handle_zero()

     if not i % 10:
         self.handle_mulTiple_of_ten()
```

1.9 过时的语言特性

> **Tip**
>
> 尽可能使用字符串方法取代字符串模块。使用函数调用语法取代 `apply()`。使用列表推导，`for` 循环取代 `filter()`，`map()` 以及 `reduce()`。

定义：
当前版本的 Python 提供了大家通常更喜欢的替代品。

结论：
我们不使用不支持这些特性的 Python 版本，所以没理由不用新的方式。

```
Yes: words = foo.split(':')

     [x[1] for x in my_list if x[2] == 5]

     map(math.sqrt, data)

     fn(*args, **kwargs)
```

```
No:  words = string.split(foo, ':')

     map(lambda x: x[1], filter(lambda x: x[2] == 5, my_list))

     apply(fn, args, kwargs)
```

1.10 威力过大的特性

> **Tip**
>
> 避免使用这些特性。

定义：
Python 是一种异常灵活的语言，它为你提供了很多花哨的特性。诸如元类、字节码访问、任意编译、动态继承、对象父类重定义、导入黑客、反射、系统内修改等等。

优点：
强大的语言特性，能让你的代码更紧凑。

缺点：
使用这些很”酷“的特性十分诱人，但不是绝对必要。使用新特性的代码将更加难以阅读和调试。开始可能还好（对原作者而言），但当你回顾代码，它们可能会比那些稍长一点但是很直接的代码更加难以理解。

结论：
在你的代码中避免这些特性。

2. Python 风格规范

2.1 分号

> **Tip**
>
> 不要在行尾加分号，也不要用分号将两条命令放在同一行。

2.2 行长度

> **Tip**
>
> 每行不超过 80 个字符。

例外：

- 长的导入模块语句。
- 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。

```
Yes: 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 = ('This will build a very long long '
     'long long long long long long string')
```

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/conten
t/v2.0/csv_file_name_extension_full_specification.html
```

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

2.3 括号

> **Tip**
>
> 宁缺毋滥地使用括号。

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

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

2.4 缩进

> **Tip**
>
> 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，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:   # Aligned with opening delimiter.
       foo = long_function_name(var_one, var_two,
                                var_three, var_four)

       # Aligned with opening delimiter in a dictionary.
       foo = {
           long_dictionary_key: value1 +
                                value2,
           ...
       }

       # 4-space hanging indent, nothing on first line.
       foo = long_function_name(
           var_one, var_two, var_three,
           var_four)
```

```
No:    # Stuff on first line forbidden.
       foo = long_function_name(var_one, var_two,
           var_three, var_four)

       # 2-space hanging indent forbidden.
       foo = long_function_name(
         var_one, var_two, var_three,
         var_four)
       }
```

2.5 空行

> **Tip**
>
> 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。

2.6 空格

> **Tip**
>
> 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。

```
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  # comment
     long_name = 2  # comment that should not be aligned

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

```
No:
     foo       = 1000  # comment
     long_name = 2     # comment that should not be aligned

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

2.7 Shebang

> **Tip**
>
> 程序第一行应该以 `#!/usr/bin/python2.7` 开始。Python 版本号具体以安装在你计算机里的为准。

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 `#!`。

2.8 注释

> **Tip**
>
> 确保对模块、函数、方法和行内注释使用正确的风格。

文件头注释

> 每个 Python 文件的头部都需要添加文件头注释。
>
> - 版权声明。
> - 开发作者。
> - 开发时间.

举个例子：

```
"""
 *********************************************************************
 *  Software License Agreement (BSD License)
 *
 *  Copyright (c) 2016, Team-Xmbot-Service-Robot
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of the Team-Xmbot-Service-Robot nor the names
 *     of its contributors may be used to endorse or promote products
 *     derived from this software without specific prior written
 *     permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTOR
S
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIREC
T,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDIN
G,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *********************************************************************
"""

# Authors: myyerrol
# Created: 2016.4.15
```

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。我们对文档字符串的惯例是使用三重双引号 `"""`。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证。根据项目使用的许可（BSD）。

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：

- 外部不可见。
- 非常短小。
- 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：
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 instanc
e
    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 tabl
e row
            to fetch.
        other_silly_variable: Another optional variable, that has a m
uch
            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 dictionar
y,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table objec
t.
    """
    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，那么文档中应该有一个属性段，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 if 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 occ
urs
# the next element is i+1
```

2.9 字符串

> **Tip**
>
> 即使参数都是字符串，使用 `%` 操作符或者格式化方法格式化字符串。不过也不能一概而论，你需要在 `+` 和 `%` 之间好好判定。

```
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_n
ame))
     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, fir
st_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。

```
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.
  """
```

2.10 TODO注释

> **Tip**
>
> 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 请求时，就移除这些代码“）。

2.11 导入格式

> **Tip**
>
> 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
```

2.12 语句

> **Tip**
>
> 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)
```

2.13 访问控制

> **Tip**
>
> 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。

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。

2.14 命名

> **Tip**
>
> 按照以下的规范命名。

应该避免的名称

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

命名约定

> - 所谓"内部"表示仅模块内可用，或者在类内是保护或私有的。
> - 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）。
> - 用双下划线 `__` 开头的实例变量或方法表示类内私有。
> - 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。
> - 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	

2.15 Main

> **Tip**
>
> 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。

3. 结语

请务必保持代码的一致性。

如果你正在编辑代码，花几分钟看一下周边代码，然后决定风格。如果它们在所有的算术操作符两边都使用空格，那么你也应该这样做。如果它们的注释都用标记包围起来，那么你的注释也要这样。

制定风格指南的目的在于让代码有规可循，这样人们就可以专注于”你在说什么“，而不是”你在怎么说“。我们在这里给出的是全局的规范，但是本地的规范同样重要。如果你加到一个文件里的代码和原有代码大相径庭，它会让读者不知所措。避免这种情况。