**Python代码规范**

**目录**

Python代码规范

目录

1. Python风格规范
   1. 分号
   2. 行长度
   3. 括号
   4. 字符串
   5. Main
2. 命名规范

2.1类

2.2函数&方法

2.3函数和方法的参数

2.4全局变量

2.5变量

2.6常量

2.7异量

2.8文件名

2.9包

2.10模块

2.11缩写

2.12前导后缀下划线

2.13特定命名方式

1. 排版规则

3.1注释

3.2缩进

3.3语句

3.4导入格式

3.5空格

1. 语言规范

4.1异常

4.2嵌套/局部/内部类或函数

4.3列表推导(List Comprehensions)

4.4默认迭代器和操作符

4.5Lambda函数

4.6默认参数值

4.7True/False的求值

4.8函数与方法装饰器

4.9避免威力过大的特性

1. **Python风格规范**

**1.1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.

**1.2行长度**

**\***每行不超过80个字符

\*

|  |  |
| --- | --- |
| Python会将 [圆括号, 中括号和花括号中的行隐式的连接起来](http://docs.python.org/2/reference/lexical_analysis.html" \l "implicit-line-joining) , 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/content/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* |

**1.3括号**

**\***宁缺毋滥的使用括号

\*

|  |
| --- |
| 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) |

**1.4字符串**

**\***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) |

**1.5Main**

**\***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.

\*

|  |
| --- |
| **def** main():  ...  **if** \_\_name\_\_ == '\_\_main\_\_':  main() |

1. **命名规则**

**2.1**类****

**\***总是使用首字母大写单词串。如MyClass。内部类可以使用额外的前导下划线。

**2.2函数&方法**

\*小写+下划线

\*注意\*：混合大小写仅被允许用于这种风格已经占据优势的时候，以便保持向后兼容。

**2.3函数和方法的参数**  
 \*如果一个函数的参数名称和保留的关键字冲突，通常使用一个后缀下划线

**2.4全局变量**  
 \*对于from M import \*导入语句，如果想阻止导入模块内的全局变量可以使用旧有 的规范，在全局变量上加一个前导的下划线。  
 \*注意\*:应避免使用全局变量

**2.5变量**

\*小写，由下划线连接各个单词。如color = WHITE，this\_is\_a\_variable = 1  
 \*注意\*：  
 1.不论是类成员变量还是全局变量，均不使用 m 或 g 前缀。  
 2.私有类成员使用单一下划线前缀标识。  
 3.变量名不应带有类型信息，因为Python是动态类型语言。如 iValue、 names\_list、dict\_obj 等都是不好的命名。

**2.6常量**  
 \*常量名所有字母大写，由下划线连接各个单词如MAX\_OVERFLOW，TOTAL。

**2.7异常**  
 \*以“Error”作为后缀。

**2.8文件名**  
 \*全小写,可使用下划线

**2.9包**  
 \*应该是简短的、小写的名字。如果下划线可以改善可读性可以加入。如mypackage。

**2.10模块**  
 \*与包的规范同。如mymodule。

**2.11缩写**  
 \*命名应当尽量使用全拼写的单词，缩写的情况有如下两种：  
 1.常用的缩写，如XML、ID等，在命名时也应只大写首字母，如XmlParser。  
 2.命名中含有长单词，对某个单词进行缩写。这时应使用约定成俗的缩写方式。

\*例如：

|  |
| --- |
| function 缩写为 fn  text 缩写为 txt  object 缩写为 obj  count 缩写为 cnt  number 缩写为 num，等。 |

**2.12**前导后缀下划线****  
 \*一个前导下划线：表示非公有。  
 \*一个后缀下划线：避免关键字冲突。  
 \*两个前导下划线：当命名一个类属性引起名称冲突时使用。  
 \*两个前导和后缀下划线：“魔”（有特殊用图）对象或者属性，例如\_\_init\_\_或者 \_\_file\_\_。绝对不要创造这样的名字，而只是使用它们。  
 \*注意\*：关于下划线的使用存在一些争议。

**2.13特定命名方式**  
 \*主要是指 \_\_xxx\_\_ 形式的系统保留字命名法。项目中也可以使用这种命名，它的意 义在于这种形式的变量是只读的，这种形式的类成员函数尽量不要重载。如

|  |
| --- |
| class Base(object): def \_\_init\_\_(self, id, parent = None): self.\_\_id\_\_ = id self.\_\_parent\_\_ = parent def \_\_message\_\_(self, msgid): 其中 \_\_id\_\_、\_\_parent\_\_ 和 \_\_message\_\_ 都采用了系统保留字命名法。 |

**3.排版规则**

**3.1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](http://www.python.org/dev/peps/pep-0257/) ). 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. |

\*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.

**3.2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 [行长度](https://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/" \l "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: *# 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)  *# 4-space hanging indent in a dictionary*  foo = {  long\_dictionary\_key:  long\_dictionary\_value,  ...  } |
| 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)  *# No hanging indent in a dictionary*  foo = {  long\_dictionary\_key:  long\_dictionary\_value,  ...  } |

**3.3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) |

**3.4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** **foofrom** **foo**  **import** bar**from** **foo.bar**  **import** baz**from** **foo.bar**  **import** Quux**from** **Foob**  **import** ar |

**3.5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.

|  |
| --- |
| 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)  Yes: dict['key'] = list[index] |
| no: spam (1)  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,  } |

**4.语言规范**

**4.1异常**

\*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:

1.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" ).

2.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** |

3.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.

4.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.

5.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.

6.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** |

**4.2嵌套/局部/内部类或函数**

\*鼓励使用嵌套/本地/内部类或函数

**定义:**

类可以定义在方法, 函数或者类中. 函数可以定义在方法或函数中. 封闭区间中定义的变量对嵌套函数是只读的.

**优点:**

允许定义仅用于有效范围的工具类和函数.

**缺点:**

嵌套类或局部类的实例不能序列化(pickled).

**结论:**

推荐使用.

**4.3列表推导(List Comprehensions)**

\*可以在简单情况下使用

**定义:**

列表推导(list comprehensions)与生成器表达式(generator expression)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) |

**4.4默认迭代器和操作符**

\*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(in和not in)

**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(): ... |

**4.5Lambda函数**

\*适用于单行函数

**定义:**

与语句相反, lambda在一个表达式中定义匿名函数. 常用于为 map() 和 filter() 之类的高阶函数定义回调函数或者操作符.

**优点:**

方便.

**缺点:**

比本地函数更难阅读和调试. 没有函数名意味着堆栈跟踪更难理解. 由于lambda函数通常只包含一个表达式, 因此其表达能力有限.

**结论:**

适用于单行函数. 如果代码超过60-80个字符, 最好还是定义成常规(嵌套)函数.

对于常见的操作符，例如乘法操作符，使用 operator 模块中的函数以代替lambda函数. 例如, 推荐使用 operator.mul , 而不是 lambda x, y: x \* y .

**4.6默认参数值**

\*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 parsed...*  ... |

**4.7True/False的求值**

\*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:

1. 永远不要用==或者!=来比较单件, 比如None. 使用is或者is not.
2. 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!
3. 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.
4. 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.
5. 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. 注意‘0’(字符串)会被当做true.

**4.8函数与方法装饰器**

**\***如果好处很显然, 就明智而谨慎的使用装饰器

**定义:**

[用于函数及方法的装饰器](https://docs.python.org/release/2.4.3/whatsnew/node6.html) (也就是@标记). 最常见的装饰器是@classmethod 和@staticmethod, 用于将常规函数转换成类方法或静态方法. 不过, 装饰器语法也允许用户自定义装饰器. 特别地, 对于某个函数 my\_decorator , 下面的两段代码是等效的:

|  |
| --- |
| **class** **C**(object):  **@my\_decorator**  **def** method(self):  *# method body ...* |
| **class** **C**(object):  **def** method(self):  *# method body ...*  method = my\_decorator(method) |

**优点:**

优雅的在函数上指定一些转换. 该转换可能减少一些重复代码, 保持已有函数不变(enforce invariants), 等.

**缺点:**

装饰器可以在函数的参数或返回值上执行任何操作, 这可能导致让人惊异的隐藏行为. 而且, 装饰器在导入时执行. 从装饰器代码的失败中恢复更加不可能.

**结论:**

如果好处很显然, 就明智而谨慎的使用装饰器. 装饰器应该遵守和函数一样的导入和命名规则. 装饰器的python文档应该清晰的说明该函数是一个装饰器. 请为装饰器编写单元测试.

避免装饰器自身对外界的依赖(即不要依赖于文件, socket, 数据库连接等), 因为装饰器运行时这些资源可能不可用(由 pydoc 或其它工具导入). 应该保证一个用有效参数调用的装饰器在所有情况下都是成功的.

装饰器是一种特殊形式的”顶级代码”. 参考后面关于 [Main](https://zh-google-styleguide.readthedocs.io/en/latest/google-python-styleguide/python_style_rules/" \l "main) 的话题.

**4.9避免威力过大的特性**

**\***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(metaclasses), 字节码访问, 任意编译(on-the-fly compilation), 动态继承, 对象父类重定义(object reparenting), 导入黑客(import hacks), 反射, 系统内修改(modification of system internals), 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. 使用奇技淫巧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.