中软国际公司内部技术规范

Python 语言编程规范

中软国际科技服务有限公司

修订声明

参考: 业界 py 规范指南，华为 Python 编码规范等。

日期	修订版本	修订描述	作者
2014-06-26	1.00	整理创建初稿	周智鹏+zWX164704
2014-07-15	1.01	格式整理	薛燕萍+xWX158728
2014-08-21	1.02	规则 5-3 修改 No 示例的代码缩 进 5 可读性 中修改重名规则名	刘萍+lWX158721
2014-09-11	1.03	第二章部分规则添加示例, 建议 2-1 中换行 示例修改, 2.5 节编码风格 添加 coding 的规 则	薛燕萍+xWX158728
2014-09-12	1.04	1、全文示例代码改为小五号字体； 2、建议 2-1 中“操作符要放行首”改成“操 作 符要放行尾”，并把“尾”加粗； 3、调整规则 2-21、建议 2-1 位置； 4、规则 2-5，添加"*"的特殊情况描述； 5、规则 2-25，添加示例和说明； 6、规则 2-28，语句细化。	周智鹏 +zWX164704 屈一 龙+qKF38450 薛燕 萍+xWX158728
2014-09-12	1.05	第八章： 1、“规则 8-6”反置 YES 和 NO 的示例实内容 一致。 2、去掉“建议 8-5”，其与规则 8-2 重复,后 边序号递减。 3、对原则 8-2 到 8-7 补充了说明内容。去掉 冗 余的原则 8-8。 4、原则 8-5 与原则 8-6 合一，原则 8-7 序号 递 减为原则 8-6。	李飞+Lwx193132
2017-08-02	1.1	统一调整格式	陈丽佳+cwx435329

目录

总则 4
1 排版 5
1.1 空格添加规则 5
1.2 空行添加规则 7
1.3 语句书写规则 7
1.4 模块导入 8
1.5 编码风格 9
2 注释 12
3 命名规范 14
4 可读性 17
5 函数 21
5.1 基本规则 21
5.2 关于对象方法 24
5.3 关于异常处理 25
5.4 函数编写时的建议 26
6 程序效率 34
7 质量保证 37

总则

★**原则**——编程时必须坚持的指导思想。

★**规则**——编程时强制必须遵守的约定。

★**建议**——编程时必须加以考虑的约定。

★**说明**——对原则、规则或建议进行必要的解释。

原则 1-1：本文档中的内容，若与 PEP 8 冲突，则以 PEP 8 为准。

说明：本文档严格基于 PEP 8 标准（原文地址：http://legacy.python.org/dev/peps/pep-0008/）编写，本文任何内容如与该标准不符，皆可被认为是编辑错误或版本陈旧导致，若发现请及

时联系我们，非常感谢！

原则 1-2：任何规则或建议均可被打破，但应当给出足以支持打破它的理由。

说明：一切规则和建议都是为了使代码更简明，更高效服务的，任何时候不能舍本逐末。

例：当应用某规则后，代码变得更晦涩，更低效，则应该摒弃之。

原则 1-3：在某些特定场景下，不符合本文档的内容也是可容忍的。

说明：通常情况下，只有建议是可容忍不符合的，而规则一般不可被容忍。

例：为了与已有代码风格保持一致，而不服从某项建议，应被看做是一种可被容忍的情况。

原则 1-4：清晰第一。

说明：清晰性是易于维护、易于重构的程序必需具备的特征。应当认为代码首先是给人看的，其次才是给机器看的。

例：代码效率与代码易读性发生冲突时，除非效率已经成为问题瓶颈，否则应优先考虑代码易读性。

原则 1-5：力求简洁。

说明：能用一行代码实现的功能，不应使用两行代码实现。

1 排版

1.1 空格添加规则

原则 2-1：在两个以上的关键字、变量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；关系密切的立即操作符（如．）除外。

说明：采用这种松散方式编写代码的目的是使代码更加清晰，在已经非常清晰的语句中没有必要再留空格，如多重括号间不必加空格。

例：

```
A = B

list.sort()
```

规则 2-1：程序中禁止 Tab 与空格混用。

说明：用 4 个空格缩进代码块，不要用 tab，特别不要混用 tab 和空格。如果可能，请设置你的 IDE 让 Tab 键自动转为 4 个空格。以免用不同的编辑器阅读程序时，因 Tab 键所设置的空格数 目不同而造成程序布局不整齐，引发错误。

规则 2-2：逗号、分号（假如用到的话）、冒号只在后面加空格。

例：

```
print a, b, c
```

规则 2-3：双目操作符（">"、">="、"<"、"<="、"=="、"+"、"-"、"%"、and、or 等）的前后加空格。

例：

```
if current_time >= MAX_TIME_VALUE:

a = b + c

 a += 2
```

规则 2-4：等号（"="）用于指名参数或默认参数值时，两边不要加空格。

例：

```
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)
```

规则 2-5："*"、"**"、"." 等操作符前后不加空格。其中"*"作为乘法操作符除外。

规则 2-6：if、for、while 后如果有括号，应当与后面的括号间加空格隔开。

例：

```
if ((a > 0 and a >= b and a > c) or (a < 0)):
```

规则 2-7：在圆括号、方括号、大括号里面不要加空格。

例：

Yes:

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

No:

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

规则 2-8：在表示参数、列表、下标、分块开始的圆括号/方括号前面不要加空格。

例：

Yes：

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

No:

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

规则 2-9：被调用函数名，开始的括号前，不需要加空格。

例：

Yes:

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

No:

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

规则 2-10：不要在赋值(或其他)操作符的前后加入多于一个空格符。 **例：**

Yes：x = 1

No ： x = 1

1.2 空行添加规则

空行可以使程序结构比较清晰。逻辑上关系比较紧密的代码放在一起，逻辑上相对比较独立的部分用空行隔开。

规则 2-11：相对独立的程序块之间必须加空行。

规则 2-12：定义类中的方法时使用空行来分隔方法。

规则 2-13：第一个方法定义和类定义本身（包括注释）之间要加一行空行。

规则 2-14：在顶级定义（可以是函数或者类定义）之间加两个空行。

规则 2-15：在 doc 字符串和它后面的代码之间加一个空行。

规则 2-16：在文件最后总是加一个空行。

说明：在文件最后总是加一个空行,可以避免很多 diff 工具生成“Nonewlineatend of file”信息。

1.3 语句书写规则

规则 2-17：语句长度>80 字符时要分成多行书写。

规则 2-18：首选使用括号（包括{},[],()）内的行延续，推荐使用反斜杠（\）进行断行。

规则 2-19：长表达式要在低优先级操作符处划分新行，操作符放在新行之首。

规则 2-20：划分出的新行要进行适当的缩进，使排版整齐，语句可读。

规则 2-21：若函数或过程中的参数较长，则要进行适当的划分。

建议 2-1：所有行限制在最大 79 字符，对顺序排放的大块文本(如文档字符串或注释)，推荐将长度限制在 72 字符。

说明：长语句分多行书写比较美观。在低优先级操作符处划分新行，可使每一行具有相对独立而完整的含义，从而比较清晰。折行时，操作符要放行尾。

例：

```
class Rectangle  (Blob):
    def    init    (self, width, height, color, emphasis, highlight):
        if width == 0 and height  == 0 and \
                color == 'red' and emphasis == 'strong' or \ #注意缩进

                highlight > 100:

            raise ValueError, "sorry，  you lose"
        if width == 0 and height  == 0 and \

                (color == 'red' or emphasis is None): #注意缩进 raise

            ValueError,  "I don't think so"

        Blob.   init    (self, width, height, color, emphasis, highlight)

    def OutputCollectItem (self,  strOutputFilePath,
                           strDeviceType, strDevVer):   #注意缩进

        pass
```

规则 2-22：一行只写一条语句。

说明：不允许把多个短语句写在一行中，if、for、do、while 等语句自占一行。

例：

Yes：

```
rect.length = 0

rect.width = 0

if pUserCR == None:

pass
```

No：

```
rect.length = 0; rect.width = 0;

if pUserCR == None:  pass
```

1.4 模块导入

规则 2-23：加载模块应分开每个模块占一行。

例：

Yes：

```
import sys
import os
```

No ：

```
import sys, os
```

建议 2-2：加载模块总是放在文件开头。

规则 2-24：按照从最常用到最不常用的顺序分组放置：

标准库导入

第三方库导入

应用程序特有的内容导入

规则 2-25：确保所有的 Python 源代码文件都是可导入的。

说明：顶级缩进中存在可执行代码,在导入时会被执行,这样的模块在被外界导入时无法明确 其状态和行为。这样的模块认为是不可导入的。通常将这部分代码放于 if name == ' main ':内，用于测试等其他目的。

例：

```
if name == ' main ':
    # 参数解析
    main()
```

建议 2-3：按照如下顺序，每组按照字母顺序排序：

以 import ... 开头的标准库和第三方库空行

所有以 from 开头的导入 空行

所有以 import 开头的导入

1.5 编码风格

规则 2-26：保持代码总体风格的一致性。

说明：如果你在编写代码，花几分钟看看你周围的代码并且弄清它的风格。如果你加入文件

中的代码看起来和里面已经有的代码截然不同，那在读者读它的时候就会被破坏节奏。尽量避免这样。

对于项目中已有的代码，可能因为历史遗留原因不符合本规范，应当看作可以容忍的特例，允许存在；但不应在新的代码中延续旧的风格。

对于第三方模块，可能不符合本规范，也应看作可以容忍的特例，允许存在；但不应在新的代码中使用第三方模块的风格。

规则 2-27：py 文件的结构按如下顺序书写：

1、起始行(Unix)

2、模块文档

3、 模块导入

4、 变量定义

5、 类定义

6、函数定义

7、主程序

规则 2-28：如果模块有编码指示说明，一般放于脚本开头，起始行(Unix)之后。同一个项目中所有 py 文件的 coding 应该保持一致，以防产生乱码。

例：

一个 UTF-8 编码的文件在脚本开头这样指示:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

2 注释

规则 3-1：注释必须和代码保持一致，代码修改后注意注释一定同步修改。

规则 3-2：块注释和代码块保持同样的缩进，#后面跟一个空格。注释放在要注释的代码前 面，紧贴代码。

例：

代码块 A

（一个空行分开独立的代码块）

代码块 B 的注释（紧贴要注释的代码，并对齐）

代码块 B

规则 3-3：行后的注释，至少离代码 2 个空格。#后面跟一个空格，这种注释风格尽量少用。

例：

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

规则 3-4：函数头注释风格，项目组内保持一致，下面给出一些参考：

例一：

```
def outputContents(args):

    #============================================================

    #功能：

    #入参：

    #出参：

    #返回值：

    #============================================================

    pass
```

例二：

```
def outputContents(args):

    '''************************************

    '入参：
```

'出参：

'功能：

'返回值：

***'''

pass

3 命名规范

原则 4-1：总体原则，新编代码必须按下面命名风格进行，现有库的编码尽量保持风格。

规则 4-1：禁止使用字符 l(小写字母 el)、O(大写字母 oh)、或 I(大写字母 eye)作为单字符的变量名；因为一些字体中，上诉字母和数字很难区分。

规则 4-2：模块命名尽量短小，使用全部小写的方式，可以使用下划线。
说明：模块名应该使用尽可能短、全小写的命名，可以在模块命名时使用下划线以增强可读性，模块名就是 py 文件名；单词之间用_分割，如： ad_stats.py。

规则 4-3：包命名尽量短小，使用全部小写的方式，不可以使用下划线。
说明：包（Package）命名应该采用全部小写，并且也要尽可能短的命名，但不允许使用下划线。因为当一个用 C 或 C++ 写的扩展模块，有一个伴随的 Python 模块来提供一个 更高层(例如，更面向对象)的接口时，C/C++模块名有一个前导下划线(如：_socket)。

规则 4-4：类的命名使用 CapWords 的方式，模块内部使用的类采用_CapWords 的方式。
说明：CapWords 这样命名是因为可从字母的大小写分出单词，类名一般是一类对象的抽象，一般以名词结束。一些常用的类命名：

类型	类名尾	示例
单例类	Singleton	DevMgrSingleton
工厂类	Factory	DevMgrFactory
对话框类	Dlg	AddDevDlg
控件类	Ctrl	DevTreeCtrl
管理类	Mgr	DevGroupMgr

规则 4-5：异常命名使用 CapWords+Error 后缀的方式。
说明：异常类名约定也服从于类命名，只是命名异常名时应使用 Error 后缀。

规则 4-6：全局变量尽量只在模块内有效，类似 C 语言中的 static。
说明：

1、全局变量，如果只在本模块内部使用，则以一个下划线开头。

2、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在全局变量上加一个前导的下划线。

3、注意:应尽量避免使用全局变量。

规则 4-7：函数命名使用全部小写的方式，可以使用下划线。
说明：函数或变量命名，若仅是一个单词，全部字母小写；若多个单词，单词全小写，中间使用“_”连接；比如： exit()、exit_app() 。

建议 4-1：不使用驼峰命名法；比如 exitApp( )。

规则 4-8：常量命名使用全部大写的方式，可以使用下划线。
说明：常量名所有字母大写，由下划线连接各个单词如 MAX_OVERFLOW、TOTAL。

规则 4-9：类的属性（方法和变量）命名使用全部小写的方式，可以使用下划线。
说明：

1、 类的属性有 3 种作用域 public、non-public 和 subclass API，可以理成 C++中的 public、private、protected，non-public 属性前，前缀一条下划线。方法开头一个下划线，类似于 protected；方法开头 2 个下划线，类似于 private；方法开头没有下划线，就 是 public。

2、 类的属性若与关键字名字冲突，后缀一下划线，尽量不要使用缩略等其他方式。

3、 为避免与子类属性命名冲突，在类的一些属性前，前缀两条下划线。比如：类 Foo 中声明 a,访问时，只能通过 Foo._Foo a，避免歧义。如果子类也叫 Foo，那就 无能为力了。

4、类的方法第一个参数必须是 self，而静态方法第一个参数必须是 cls。

规则 4-10：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。
说明：

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：

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。

规则 4-11：自定义标识符避免使用 python 关键字，如：object，str，id，sort 等。

规则 4-12：自定义方法名应避免使用内建方法名格式，如： xxx ()。

说明：尽量避免用下划线作为变量名的开始；下划线对解释器有特殊的意义，而且是内建标识符所使用的符号，程序员避免用下划线作为变量名的开始。一般来讲，变量名_xxx 被看作是“私有的”，在模块或类外不可以使用。当变量是私有的时候，用_xxx 来表 示变量是很好的习惯。

4 可读性

原则 5-1：任何人都能写出计算机可以理解的代码，唯有写出人类容易理解的代码，才是优秀的程序员。

规则 5-1：注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

例：

Yes：

```
if width == 0 and height == 0 and (color == 'red' or emphasis is None): raise ValueError, "I don't think so"
```

No：

```
if width == 0 and height  == 0 and color == 'red' or emphasis is None:
    raise ValueError, "I don't think so"
```

说明：

1、 防止因默认的优先级与设计思维不符而导致程序出错，防止阅读程序时产生误解。

2、 多个运算符进行运算时，即使运算符的优先顺序可以保证逻辑正确，也要求使用括号，以保证逻辑和代码清晰。

规则 5-2：避免使用不易理解的数字，用有意义的标志来替代。涉及物理状态或者有物理意义的数值，不应直接使用数字，必须用有意义的变量来代替。

例：

Yes：

```
TRUNK_IDLE = 0
TRUNK_BUSY= 1
if Trunk[index].trunk_state == TRUNK_IDLE:Trunk[index].trunk_state = TRUNK_BUSY
 # program code
```

No：

```
if Trunk[index].trunk_state == 0:Trunk[index].trunk_state = 1
 # program code
```

说明：代码中直接出现的数字，怎样区分到底是否应该用变量来代替？ 一般的原则是：

1、 如果此数字具有明确的物理状态或物理意义，必须使用变量进行替换。这样可以使代码的可读性大大增强。

2、 如果此数字被多处使用，则也应该使用变量进行替换。这样的好处是，如果以后需要修改这个值，只需将变量定义的地方修改了就可以了，维护的工作量大大减少。

3、 另外，对于写日志、界面等输出的字符串，也建议使用变量定义。这样，在程序需 要修改日志、界面等输出时，只需修改定义一个地方就可以了。比如国际化时

就有 这样的问题。

规则 5-3：长语句用续行符换行后多加一个缩进，使维护人员读代码时看到代码首行即可判定这里存在换行。

例：

Yes：

```
if color == WHITE or color ==BLACK \
    or color == BLUE:
    do_something(color)
```

No：

```
if color == WHITE or color ==BLACK \
 or color == BLUE:
    do_something(color)
```

规则 5-4：如果代码行在长度超过了 80 个字符，或使用了多层函数调用（例如：map(lambda x:x[1], filter(...))），那这就是一个信号提醒你最好重新写一个普通的循环。

例：

Yes：

```
[x(1) for x in my_list if x[2] == 5]
```

No：

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

说明：实用的函数式编程使代码更加紧凑，但是高阶函数式编程恐怕更难理解。

规则 5-5：避免使用复杂的列表解析和生成器表达式，而改用循环。

例：

Yes：

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

for x in range(5):
    for y in range(5):
        if x * y > 10:
            for z in range(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)
```

No：

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


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)
```

说明：简单的列表解析可以比其他的列表创建方法更加清晰简单，生成器表达式也可以十分高效；但复杂的列表解析或者生成器表达式可能更加难以阅读。

规则 5-6：较复杂的正则表达式，使用 re.VERBOSE，增强代码可读性。

例：

Yes：

```
Charref = re.compile(r"""   &[#] # Start of a numeric entity reference (
        [0-9]+[^0-9]                    # Decimal form
        | 0[0-7]+[^0-7]                 # Octal form
        | x[0-9a-fA-F]+[^ 0-9a-fA-F]    #Hexadecimal form
        )
        """, re.VERBOSE)
```

No：

```
Charref = re.compile("""&[#]([0-9]+[^0-9] | 0[0-7]+[^0-7] | x[0-9a-fA-F]+[^ 0-9a-fA-F])""")
```

规则 5-7：避免使用难以理解的高级特性。

说明：Python 是一种极为灵活的语言，提供诸如 metaclass、 bytecode 访问、即时编译、动态继 承、object reparenting、import hacks、反射、修改系统内部结构等很多很炫的特性。

使用它们能使你的代码更紧凑，而且很“酷”，但使用了这些并不常见的特性的代码会 更难读、更难懂、更难 debug。也许在刚开始的时候好像还没这么糟（对代码的原作者来说），但当你重新回到这些代码，就会觉得这比长点但简单直接的代码要更难搞。因此原则上应避免使用这些特性。

建议 5-1：关系较为紧密的代码应尽可能相邻。

例：

Yes：

```
rect.length = 10
rect.width = 5      # 矩形的长与宽关系较密切，放在一起
```

```
char_poi = stra No：
rect.length = 10 char_poi = stra rect.width = 5
```

说明：便于程序阅读和查找。

建议 5-2：不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

5 函数

5.1 基本规则

规则 6-1： 对所调用函数的错误返回值要仔细、全面地处理。

规则 6-2： 明确函数功能，精确（而不是近似）地实现函数设计。

规则 6-3： 定义函数默认参数时候的规则

给函数定义默认参数可以让调用者不必每次都传入一堆不必要的入参，很方便；此外由于 Python 不支持方法/函数重载，因此参数默认值也是“模仿”重载行为的一种简单方法。

但是，默认参数值会在模块加载的时候被赋值，如果参数是一个列表或者字典之类的可变对象就有可能造成问题。如果函数修改了这个对象（比如在列表最后附加了一个新的项），那默认值也就改变了。

例：

```
>>>def fun(v, l=[])
    l.append(v)print l
>>>fun(1)[1]
>>>fun(2)[1, 2]
>>>fun(3, []) [3]
```

因此在定义默认参数的时候要遵守如下规则：

1、 规定：不要把可变对象当作函数或方法定义的默认值。

Yes：

```
def foo(x, y=None): if y is None:
        y = []
        y.append(x)
```

No：

```
def foo(x, y=[]):
    y.append(x)
```

2、 建议：调用函数的代码必须对默认参数使用指名赋值。这多少可以帮助代码的文档化，并且当增加新参数进来时帮助避免和发现破坏原有接口。

例：

```
def foo(x, y=1): z = x + y
  print z Yes：
   foo(1) foo(1, y=2)
No：
   foo(1, 2)
```

规则 6-4： 编写可重入函数时，对于使用的全局变量，可以使用信号量（即 P、V 操作）等手段对其加以保护。

说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个线程（进程）调用此函数时，很有可能使有关全局变量变为不可知状态。

例：假设 Exam 是整型全局 变量，函数 Squre_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

```
def example(self, para):
    global  Exam
    Exam = para#**

    temp = Square_Exam()
    return temp
```

此函数若被多个线程（进程）调用的话，其结果可能是未知的，因为当（**）语句刚执行完后，另外一个使用本函数的进程可能正好被激活，那么当新激活的进程执行到此函数时，将使 Exam 赋予另一个不同的 para 值，所以当控制重新回到“temp = Square_Exam( )”后，计算出的 temp 很可能不是预想中的结果。此函数应如下改进。

```
def example(self, para):
    global  Exam
    Exam = para#**

     [审请信号量操作]
     temp = Square_Exam()
     [释放信号量操作]

     # 其他线程（进程）必须等待本进程释放信号量后，才能再用本信号
    return temp
```

规则 6-5： 在同一项目组应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责，缺省是由函数调用者负责。

说明：对于模块间接口函数的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过

程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

如果一个函数仅在内部使用，则由函数调用者负责参数的合法性检查，保证传入的参数是正确的。如果函数是对外接口的函数暴露给外界来调用或者是直接处理外部传来的数据，则必须由函数本身来检查传入参数是否有效，以保证有容错能力，典型的例子是 API, COM 接口函数等。

规则 6-6： 内嵌函数里不要对母函数的变量重新赋值。内嵌函数可以引用定义在母函数中的变量，但无法对它们重新赋值。在一个代码块中，如果对一个名字重新赋值，那么 python 就将把这个名字当做局部变量，所有对它的引用都受此影 响成为对局部变量的引用，哪怕是先调用后赋值的情况。如果对它申明为全局变量，那么这个名字还是会被当做全局变量。

上述“代码块”里生效的这种特性，叫“词法域”（Lexical Scoping），以下是使用这 一特性的例子：

```
def foo(x):
    print x

def getAdder(summand1):
    """getAdder  返回把数字与一个给定数字相加的函数"""
    def anAdder(summand2):
        return summand1 + summand

    return anAdder
foo = getAdder(3) print foo(2)
```

输出：

```
5
```

这里的输出是 5，而不是 2，就是因为 foo 已经指向内嵌函数 anAdder。

规则 6-7： 函数和方法的文档字符串规则。

如果不是用途非常明显而且非常短的话，所有函数和方法都应该有文档字符串。此外，所有外部能访问的函数和方法，无论有多短、有多简单，都应该有文档字符串。 文档字符串应该包括函数能做什么、输入数据的具体描述（“Args:”）、输出数据的具体描述（“Returns:”、“Raises:”、或者“Yields:”）。 文档字符串应该能提供调用此函数相关的足够信息，而无需让使用者看函数的实现代码。 如果参数要求特定的数据类型或者设置了参数默认值，那文档字符串应该明确说明这两点。

“Raises:”部分应该列出此函数可能抛出的所有异常。生成器函数的文档字符串应该用“Yields:”而不是 Returns:。

函数和方法的文档字符串一般不应该描述实现细节，除非其中涉及非常复杂的算法。在难以理解的代码中使用块注释或行内注释会是更合适的做法。

例：

```
def fetchRows(table, keys):
    """取出表中的多行内容。
    Args:
        table: 打开的表。 Table 类的实例。
        keys: 字符串序列，表示要从表中取出的行的键值。
    Returns:

        一个字典，映射指定键值与取出的表中对应行的数据：
        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}
        如果 keys 参数中的键值没有出现在字典里，就表示对应行在表中没找到。
    Raises:
        IOError: 访问table.Table 对象时发生的错误。
    """
    pass
```

5.2 关于对象方法

规则 6-8： 决定你的属性（或方法）是否应该私有。私有和非公共（保护）的区别是：前者对继承类将绝对不会起作用，而后者将是对继承类有用的。

类私有属性应该有两个前缀下划线，没有后缀下划线。非公共属性应该仅仅有一个下划线，没有后缀下划线

例：

```
class A(object):
    # 私有方法，对继承类无效,外部不能直接调用
    def  fun1(self):
        # 私有属性，对继承类无效,外部不能直接调用
        self.  attr1 = None
    #非公有（保护）方法，不允许外部直接调用（pyt hon 对此无直接限制，需要开发人员自己保证，通常可通过 pylint 检查发现）
```

```
def _fun2(self):
    #非公有（保护）属性，不允许外部直接调用（python 对此无直接限制，需要开发人员自己保证，通常可通过 pylint 检查发现）
    self._attr2 = None
```

5.3 关于异常处理

规则 6-9： 抛异常的规则很多时候，为了处理某个错误（或其他原因）我们需要打破代码块的正常控制流（让控制流跳过多个步骤，比如一步就从 N 层嵌套函数中返回，而不必继续把错误代码一步 步执行到底），这个时候有个很 Pythonic 的方法：抛异常。

抛异常应该遵守如下规则：

1、 要像这样抛出异常：raise MyException("Error message") 或者 raise MyException，而不要用双参数的形式（raise MyException, "Error message"） 或者已废弃的基于字符串的异常（raise "Error message"）。

2、 模块和包应该定义自己的特定领域的基础异常类，而且这个类应该继承自内置的 Exception 类。

```
class Error(Exception):
    """Baseexception for all exceptions raised in module Foo."""
    Pass
```

规则 6-10： 异常处理的规则

Python 的异常捕获非常强大，你不应该利用它来掩盖你程序里的真正的问题，基于这个原则应该遵守如下规则：

1、 规定：永远不要直接捕获所有异常，也就是不要使用 except 语句，或者捕获 Exception 和 StandardError，除非你会把异常重新抛出或者是在你线程最外层的代码块中（而且你会打印一个出错信息）。因为 Python 是很健壮的，except 会真正捕获包括 Python 语法错误在内的所有错误。使用 except 很容易会掩盖真正的 Bug。

2、 规定：在 try/except 块中使代码量最小化。try 里头的内容越多，就更有可能出现你原先并未预期会抛出异常的代码抛出异常的情况。在这种情况下，try/except 代码块就隐藏了真正的错误。

3、建议：无论 try 语句块是否抛出异常都使用 finally 执行一段代码，这在做清除工作的时候经常是很有用的，比如关闭文件。

规则 6-11： 应该优先选择基于类的异常定义，而不是基于字符串的异常定义。模块或包应该在它们自己的应用范围内自行定义异常类。这些异常类应继承自内建的基本异常类。一定要加上一个类的文档字符串。如：

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

规则 6-12： 异常捕获后，如果不对该异常进行处理，则应该将 traceback.format_exc()信息记录日志。

说明：若有特殊原因必须用注释加以说明。

规则 6-13： 自己抛出的异常必须要填写详细的描述信息。

说明：便于问题定位。

规则 6-14： 数据库操作、IO 操作、锁操作结束后要关闭操作对象，如果使用了 try…except…finally 结构，在最后的 finally 块中将对象关闭。

5.4 函数编写时的建议

建议 6-1： 防止将全局变量或者类变量的引用作为函数的返回值。说明：除非是制作内部使用且完全不存在对函数返回值的改变操作，否则将全局变量或者类

变量的引用作为函数的返回值，有可能错误地改变变量内容，所以很危险。若存在对函数返回值的改变操作，或者函数是作为外部接口使用的，则应返回变量的拷贝。

例：

Yes:

```
a = [1, 2, 3]
def foo():
    return a[:]
```

No:

```
a = [1, 2, 3]
def foo():
    return a # 返回了全局变量的引用，不太好
```

建议 6-2： 避免使用全局变量。

这里全局变量指在模块级别声明的变量。由于在模块被导入时会完成模块级别变量的赋值，

所以有可能导致在导入时改变模块的行为。 因此，推荐使用类变量而避免使用全局变量。除非：

1、 作为脚本中的默认选项。

2、 模块级别常量，例如 PI = 3.14159 。

3、 有时全局变量对缓存函数返回值或其他需要的运算结果会是挺有用的。

4、 在需要的时候，全局变量应该被用作模块的内部变量，并通过公开的模块级函数来访问。

建议 6-3： 函数的规模尽量限制在 200 行以内。

说明：不包括注释和空格行。

这主要是为了结构清晰起见。“如果你要编写一段超过 200 行代码的子程序，那你就 要小心了。对于超过 200 行代码的子程序来说，没有哪项研究发现它能降低成本和/或降低出错率，而且在超过 200 行后，你迟早会在可读性方面遇到问题。”

建议 6-4： 一个函数仅完成一件功能。

例：

Yes:

```
def sum(x, y):
    return x+y

def square(x):
    return x*x
```

No:

```
def func(x, y):
    z = x + y
    return z*z
```

建议 6-5： 为简单功能编写函数。

说明：虽然为仅用一两行就可完成的功能去编函数好像没有必要，但用函数可使功能明确化，增加程序可读性，亦可方便维护、测试。

建议 6-6： 不要设计多用途面面俱到的函数。

例：

Yes:

```
def sum(x, y):
    return x+y


def multiply(x, y):
```

```
        return x*y
```

No:

```
def super_fun(x, y, flag=0):
    if 0 == flag:
        return x+y
    if 1 == flag:
        return x*y
    return None
```

说明：多功能集于一身的函数，很可能使函数的理解、测试、维护等变得困难。

建议 6-7： 函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。
例：

Yes:

```
def foo(x):
    sum = 1
    sum += x
    print sum
```

No:

```
sum = 1
def foo(x):
    sum += x
    print sum
```

建议 6-8： 尽量不要编写依赖于其他函数内部实现的函数。
说明：此条为函数独立性的基本要求。由于目前大部分高级语言都是结构化的，所以通过具体语言的语法要求与解释器功能，基本就可以防止这种情况发生。

建议 6-9： 避免设计多参数函数，不使用的参数从接口中去掉。
说明：目的减少函数间接口的复杂度。

建议 6-10： 非调度函数应减少或防止控制参数，尽量只使用数据参数。
例：

Yes:

```
def add(a, b):
    return (a + b)

def sub(a, b):
    return (a - b)
```

No:

```
def add_sub(a, b, add_sub_flg):
    if add_sub_flg == INTEGER_ADD:
        return (a + b)
    else:
        return (a - b)
```

说明：本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令，来启动相应的功能实体（即函数或过程）， 而本身并不完成具体功能。控制参数是指改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。

建议 6-11：检查函数所有参数输入的有效性。

建议 6-12：检查函数所有非参数输入的有效性，如数据文件、公共变量等。
说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输 入，即非参数输入。函数在使用输入之前，应进行必要的检查。

建议 6-13：函数名应准确描述函数的功能。

建议 6-14：使用动宾词组为执行某操作的函数命名。如果是 OOP 方法，可以只有动词（名词是对象本身）。

例：参照如下方式命名函数。

```
def print_record (self, rec_ind):

def input_record ():

def get_current_color ():
```

建议 6-15：避免使用无意义或含义不清的动词为函数命名。
说明：避免用含义不清的动词如 process、handle 等为函数命名，因为这些动词并没有说明要具体做什么。

建议 6-16：函数的返回值要清楚、明了，让使用者不容易忽视错误情况。
说明：函数的每种出错返回值的意义要清晰、明了、准确，防止使用者误用、理解错误或忽视错误返回码。

编制统一的错误码，无论是内部函数还是对外的接口函数，都使用统一编制的错误码作为返回值。尽量避免使用 boolean 作为函数的返回值，因为 boolean 类型能够表达的含义非

常有限，也不是非常清晰。

建议 6-17：除非必要，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回。

使用检查工具，如 PyLint，pycheck 等，这种问题可以全部避免。

建议 6-18：让函数在调用点显得易懂、容易理解。

建议 6-19：避免函数中不必要语句，防止程序中的垃圾代码。

说明：程序中的垃圾代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

不用的代码先注释掉，这样如果以后还用，就可以直接将注释去掉即可。如果确认这段代码完全没有用途时可以将其删除。

建议 6-20：防止把没有关联的语句放到一个函数中。

例：

Yes:

```
def Init_Rect(Rect):
    # 初始化矩形的长与宽
    Rect.length = 0
     Rect.width = 0

def Init_Point(Point) :
    # 初始化“点”的坐标
    Point.x = 10
    Point.y = 10
```

No:

```
def Init_Var(Rect, Point):
    # 初始化矩形的长与宽
    Rect.length = 0
    Rect.width = 0
    # 初始化“点”的坐标
    Point.x = 10
    Point.y = 10
```

说明：防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时

也使函数或过程的功能不明确。

建议 6-21： 如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。
说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的函数。
如果多处代码的执行流程一致，只是具体处理数据有所不同，则需要考虑将其抽象为函 数或变量。

建议 6-22： 功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。
说明：模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过小的函数，特别是扇入很低的或功能不明确的函数，不值得单独存在。

建议 6-23： 设计高扇入、合理扇出（小于 7）的函数。
说明：扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调
用它。扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，如总 是 1，表明函数的调用层次可能过多，这样不利程序阅读和函数结构的分析，并且程序运行 时会对系统资源如堆栈空间等造成压力。函数较合理的扇出（调度函数除外）通常是 3-5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函 数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的 功能，也不能违背函数间的独立性。扇入越大，表明使用此函数的上级函数越多，这样的函 数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层 函数应该有较高的扇入。较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

建议 6-24： 减少函数本身或函数间的递归调用。
说明：递归调用特别是函数间的递归调用（如 A->B->C->A），影响程序的可理解性；递归
调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应减少没必要的递归调用。

建议 6-25： 仔细分析模块的功能及性能需求，并进一步细分，同时若有必要画出有关数据流 图， 据此来进行模块的函数划分与组织。

说明：函数的划分与组织是模块的实现过程中很关键的步骤，如何划分出合理的函数结构，关系到模块的最终效率和可维护性、可测性等。根据模块的功能图或/及数据流图映射出函数结构是常用方法之一。

建议 6-26：改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。优化函数结构时，要遵守以下原则：

1、不能影响模块功能的实现。

2、仔细考查模块或函数出错处理及模块的性能要求并进行完善。

3、通过分解或合并函数来改进软件结构。

4、考查函数的规模，过大的要进行分解。

5、降低函数间接口的复杂度。

6、不同层次的函数调用要有较合理的扇入、扇出。

7、函数功能应可预测。

8、提高函数内聚。（单一功能的函数内聚最高）

说明：对初步划分后的函数结构应进行改进、优化，使之更为合理

建议 6-27：在多任务操作系统的环境下编程，要注意函数可重入性的构造。
说明：可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。另外，解释器是否提供可重入函数库，与它所服务的操作系统有关，只有操作系统是多任务时，解释器才有可能提供可重入的函数库。

建议 6-28：避免使用 BOOL 参数。
说明：原因有二，其一是 BOOL 参数值无意义，Ture/False 的含义是非常模糊的，在调用时很难知道该参数到底传达的是什么意思；其二是 BOOL 参数值不利于扩充。还有 None 也是一个无意义的单词。

建议 6-29：对于序列（string，list，tuple）长度是否为空的判断，使用”if seq”和”if not seq”。

建议 6-30：数据库、IO 及锁对象的操作，建议使用 with 操作防止这些对象未被关闭。
例：

```
lock = threading.Lock()

with lock:
```

```
# Critical section of code
    ...
```

建议 6-31： 对于提供了返回值的函数，在引用时最好使用其返回值。

建议 6-32： 避免使用 Lambda 函数 lambda 函数是仅包含表达式、没有其他语句的匿名函数，通常用于定义回调或者用于 map() 和 filter()这样高阶函数的运算符。

它的优点是方便，而缺点是比本地函数更难阅读和调试（没有名字意味着堆栈追踪信息更难理解），此外它只能包含一个表达式，因而表达能力也比较有限。因此，建议不要用 lambda 函数（通常可使用列表表达式替代）。

例：

Yes:

```
temp_lst = [x.strip() for x in lst if x.startwith('a')]
```

No:

```
temp_lst = filter(lambda x: x.startswith('a'), lst)
 temp_lst = map(lambda x: x.strip(), temp_lst)
```

6 程序效率

原则 7-1：仔细分析有关算法，并进行优化。建议尽量使用 Python 自带的内建方法。

原则 7-2：仔细考查、分析系统及模块处理输入（如事务、消息等）的方式，并加以改进。程序效率。

说明：软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。

原则 7-3：编程时，要随时留心代码效率；优化代码时，要考虑周全。

原则 7-4：不应花过多的时间拼命地提高调用不很频繁的函数代码效率。说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

原则 7-5：在多重循环中，应将最忙的循环放在最内层。
说明：减少 CPU 切入循环层的次数。
例：如下代码效率不高。

```
for row in xrange  (100):
    for col in xrange  (5):
        sum += a[row][col]
```

可以改为如下方式，以提高效率。

```
for col in xrange  (5):
    for row in xrange (100):
        sum += a[row][col]
```

原则 7-6：尽量减少循环嵌套层次。

原则 7-7：避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。
说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不 可以。
例：如下代码效率稍低。

```
for ind in range (MAX_RECT_NUMBER):if
    (data_type == RECT_AREA):
        area_sum += rect_area[ind]
    else:
```

```
            rect_length_sum += rect[ind].length
            rect_width_sum += rect[ind].width
```

因为判断语句与循环变量无关，故可如下改进，以减少判断次数。

```
    if data_type == RECT_AREA:
        for ind in xrange (MAX_RECT_NUMBER):
            area_sum += rect_area[ind]
    else:
        for ind in xrange (MAX_RECT_NUMBER):
            rect_length_sum += rect[ind].length
            rect_width_sum   += rect[ind].width
```

原则 7-8：不要一味追求紧凑的代码。

说明：因为紧凑的代码并不代表高效的机器码。

一般情况下，对于冗余的代码是需要精简的，这主要是为了清晰和维护容易起见，精简 之后的代码仍然是要清晰和易读，并且效率要保证。

规则 7-1： 编程时要经常注意代码的效率。

说明：在进行大数据量操作时，要注意代码的效率。

例：在进行大数据的查找时，如果数据类型是列表，最好将其先转换成元组，在进行查找。因为元组和列表相比其缺点是不可改变。但它有一个优点就是查询时效率要高于 列表。

规则 7-2： 在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。

说明：不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测性造成影响。

规则 7-3： 局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

规则 7-4： 通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

说明：这种方式是解决软件空间效率的根本办法。

规则 7-5： 循环体内工作量最小化，避免在循环体内创建对象，尽量在循环体外使用 if 判断 语句和 try…except…finally 结构等。

说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。

例：如下代码效率不高。

```
for ind in xrange (MAX_ADD_NUMBERJ):
    sum += ind
    back_sum = sum          # backup sum
```

语句“back_sum = sum”完全可以放在 for 语句之后，如下。

```
for ind in xrange (MAX_ADD_NUMBERJ):
    sum += ind
back_sum = sum              # backup sum
```

7 质量保证

原则 8-1：在软件设计过程中构筑软件质量。

原则 8-2：保证程序的正确性。
说明：指程序要实现设计要求的功能。

原则 8-3：保证程序的稳定性、安全性。
说明：指程序要实现的稳定、安全、可靠。

原则 8-4：保证程序的可测试性。
说明：指程序要具有良好的可测试性。

原则 8-5：保证程序的执行效率。
说明：指首先保证全局效率，其次在不影响全局效率的前提下提升局部效率。
例：优先保证软件系统的整体效率，在此基础上尽量提高某个模块/子模块/函数的本身效率。

原则 8-6：保证代码的规范/可读性。

说明：指程序书写风格、命名规则等要符合规范。

规则 8-1：过程/函数中申请的（为打开文件而使用的）文件句柄，在过程/函数退出之前要 关闭。
例：

```
Yes： f = open('myFile', 'r')
        data = [line.strip() for line in  f.readlines()]
        f.close()
No：  f = open('myFile', 'r')
        data = [line.strip() for line in f.readlines()]
```

说明：网络连接不关闭和文件句柄不关闭，是较常见的错误，而且稍不注意就有可能发生。这类错误往往会引起很严重后果，且难以定位。

规则 8-2：防止内存操作越界。
例：

```
Yes：a =['one']
    print  a[0]  No：
a = ['one']
```

```
print a[1]
```

说明：内存操作主要是指对列表、元组等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细小心。

规则 8-3：认真处理程序所能遇到的各种出错情况。

例：

Yes：

```
if language == "cn":print
    "中文语言"
elif language == "en":
    print "中文语言"
else：
    print "未知语言"
```

No：

```
if language == "cn":print
    "中文语言"
elif language == "en":
    print "中文语言"
```

说明：多分支 if 判断一定要有最后的 else 语句，以保证所有情况都能捕获到。对没有 else 分支的语句要小心对待。

规则 8-4：系统运行之初，要初始化有关变量及运行环境，防止未声明的变量被引用。对于类的成员属性，必须在 init 函数中进行初始化，防止未被声明的属性被使用。

例：

Yes：

```
count = 0
print count No：
 print count
```

规则 8-5：系统运行之初，要对加载到系统中的数据进行一致性检查。

例：

Yes：

```
def double_int(num):
    if num isinstance int:
```

```
            return num*2
        else:
            return None
```

No：

```
    def  double_int(num):
        return num*2
```

说明：使用不一致的数据，容易使系统进入混乱状态和不可知状态。

规则 8-6：严禁随意更改其它模块或系统的有关设置、配置和接口。

例：

Yes：

chn.py

```
    language = ['cn']
```

inter.py

```
    import chn
    language = chn.language[:]
    language.append('en')
```

No：

chn.py

```
    language = ['cn']
```

inter.py

```
    import chn
    language = chn.language
    language.append('en')
```

说明：编程时，不能随心所欲地更改不属于自己模块的有关设置如列表的大小等。充分了解系统的接口之后，再使用系统提供的功能。

规则 8-7：编程时，要防止差 1 错误。

例：

Yes：
```
if age >= 18:
    print "成年"
else:
    print "未成年"
```
No：
```
if age > 18:
    print "成年"
else:
    print "未成年"
```

说明：此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，由此引起的后果，很多情况下是很严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作符进行彻底检查。

规则 8-8：要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。

例：

Yes：
```
if age == 18:
    print "成年"
```
No：
```
if age = 18:
    print "成年"
```
说明：形式相近的操作符最容易引起误用。

规则 8-9：Python 中，多线程的中的子线程退出必需采用主动退出方式。

例：

Yes：
```
import
    thread
    def loop():
        print 'loop done!'
        thread.exit()
    thread.start_new_thread(loop, ())
```
No：
```
import
  thread def
  loop():
```

```
        print 'loop done!'

    thread.start_new_thread(loop, ())
```

说明：当一个线程结束计算，它就退出了。线程可以调用 thread.exit()之类的退出函数，也可以使用 Python 退出进程的标准方法，如 sys.exit()或抛出一个 SystemExit 异常等。

规则 8-10：字符串普通拼接，应该用 % 运算符来格式化字符串，即使所有的参数都是字 符串。不过你也可以在 + 和 % 之间做出你自己最明智的判断。

例：

Yes：

```
x = a + b

x = '%s, %s!' % (imperative, expletive)

x = 'name: %s; score: %d' % (name, n)
```

No：

```
x = '%s%s' % (a, b)   # 这种情况下应该用+

x = imperative + ', ' + expletive +  '!'

x = 'name: ' + name + '; score: ' + str(n)
```

规则 8-11：字符串循环拼接，应该避免在使用 + 或 += 来连续拼接字符串。因为字符串是 不变型，这会毫无必要地建立很多临时对象，从而二次方级别的运算量而不是线性运算时间。相反，应该把每个子串放到list 里面，然后在循环结束的时候用“.join()”拼接这个列表。

例：

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>'
```

规则 8-12：字符串行拼接，通常使用隐含的行连接（implicit line joining）会更清晰，因为多行字符串不符合程序其他部分的缩进风格。

例：

Yes：print ("这样会好得多。\n"
"所以应该这样用。\n")

No：print """这种风格非常恶心。
不要这么用。
"""

建议 8-1：精心地构造、划分子模块，并按“接口”部分，及“内核”部分合理地组织子模 块，以提高“内核”部分的可移植性和可重用性。

说明：对不同产品中的某个功能相同的模块，若能做到其内核部分完全或基本一致，那么无论对产品的测试、维护，还是对以后产品的升级都会有很大帮助。

建议 8-2：精心构造算法，并对其性能、效率进行测试。

建议 8-3：以-t 选项来启动 Python 命令行解释器，会就代码中混用空格和跳格的位置进行警 告信息。以-tt 选项，会升级成错误信息。强烈推荐使用这些选项。

例：

Yes： python -t demo.py
python -tt demo.py

No： python demo.py

建议 8-4：对较关键的算法最好使用其它算法来确认。

建议 8-5：为用户提供良好的接口界面，使用户能较充分地了解系统内部运行状态及有关系统出错情况。

建议 8-6：系统应具有一定的容错能力，对一些错误事件（如用户误操作等）能进行自动补救。

建议 8-7：对一些具有危险性的操作代码（如写硬盘、删数据等）要仔细考虑，并防止对数据、硬件等的安全构成危害，以提高系统的安全性。

建议 8-8：使用第三方提供的软件开发工具包或控件时，要注意以下几点：

1. 充分了解应用接口、使用环境及使用时注意事项。

2. 不能过分相信其正确性。

3. 除非必要，不要使用不熟悉的第三方工具包与控件。说明：使用工具包与控件，可加快程序开发速度，节省时间，但使用之前一定对它有较充分的了解，同时第三方工具包与控件也有可能存在问题。

建议 8-9：资源文件（多语言版本支持），如果资源是对语言敏感的，应让该资源与源代 码文件脱离，具体方法有下面几种：使用单独的资源文件、其它单独的描述文件（如数据库格式）。

说明：日志，界面显示等，是禁止直接写成“硬”代码的，必须统一规划到资源文件，以避免后期语言移植的巨大工作量。

建议 8-10：尽可能避免使用 3.X 不支持的语法。Python3.x 是大势所趋，但并非所有 2.x 的库都有 3.x 的版本了，所以两者的取舍是个令人纠结的事情。一个可行的办法是采用既支持 2.x 语法又支持 3.x 语法的 Python2.6/2.7（一般认为，如果有意，是可以写出使用 3.x 语法并运行于 2.6/2.7 的程序的）。

1、 print 在 3.x 里不再是表达式，是函数。

例：

Yes：print("fish")

No：print "fish"

2、 在 3.x 里 Unicode 字符串与 Byte 字符串不能自动转换，需要手动转换。

例：

Yes："fish"+b"panda".decode("utf-8")No：

"fish"+b"panda"

3、 在 3.x 里捕获异常的语法由“except exc, var”改为“except exc as var”

4、 使用语法 except (exc1, exc2) as var 可以同时捕获多种类型的异常。

5、 八进制数不再使用“0777”这种形式，要写成“0o777”

6、 不要比较两个没有明确顺序意义的对象，比如数字和字符串在 2.x 里，比较 1 < ""会返回 True，而在 Python 3.0 里面会抛出异常。

说明：为了让将来的切换不至于遇到很多麻烦，建议哪怕在 2.6/2.7 上写脚本，也不要使用明确规定 3.x 不再支持的语法。