python 安全编码规范

整理知识，传播智慧

好好学习，天天向上

gopher：石鸟路遇

email：1245863260@qq.com

Simple, Poetic, Pithy

文档为个人知识积累，如有不正确的地方，请指正。

目标

针对python语音编程中的代码风格要求，编码、异常处理、并发与并行、性能 和其他方面，描述可能导致程序错误、低效、难以理解维护的常见编程问题

1、排版

1.1、程序块采用4个空格缩进风格编写

程序块儿采用缩进风格编写，缩进的空格数为4个，是业界通用的标准

> 仅使用*tab*也是允许的

1.2、禁止混合使用空格和tab

推荐的缩进方式是空格，可以使用tab。如果已有代码中混合使用了空格和tab，要全部转换成空格

1.3、python文件必须使用UTF-8编码

python文件必须使用utf-8编码。可以在文件头部加上

```
# -*- coding: utf-8 -*-
# coding=utf-8   （=号两边不要空格）
```

1.4、一行只写一条语句

不允许把多个短语句写在一行中。多条语句写在一行，明显的缺点就是在调试的时候无法单步执行

```
# 反例
length = 0 ; width = 0

# 正例
length = 0
width = 0
```

1.5、相对独立的程序块之间、变量说明之后，必须加空行

独立的代码块之后加上空行，可以增强代码的理解性。

```
# 正例
if command:
    do()

writer = get()
```

1.6、行宽不超过80个字符。与python标准库看齐

一行字符太多，阅读不方便

```
# 反例
if width == 0 and height == 0 and color == "red" and emphasis == "strong" and
heihlight > 100 :
    pass

# 正例
if width == 0 \
        and height == 0  \
        and color == "red" \
        and emphasis == "strong" \
        and heihlight > 100 :
    pass
```

1.7 、两个以上的关键字、变量、常量进行对等操作时，他们之间的操作符前后要加上空格

```
# 正例
a = b + c
a -= 2

if a > 0:
```

1.8、加载模块时必须分开每个模块占一行

```
# 正例
import os
import sys

from sys import stdin, stdout
```

1.9、导入部分置于模块注释和文档字符串注释之后，模块全局变量和常量申明之前

导入时按照标准库、第三方库、本地库的顺序导入，并在几组导入直接加个空行

```
# 正例
"""
注释
"""
import sys
import os

from other import cfg

from cinder import context
```

1.10、尽量避免使用 from xxx import *的方式导入某模块的所有成员

from xxx import * 的方式导入会将所有成员挨个赋值的当前范围，如果已有变量，则会默认将其覆盖。这种方式容易导致命名冲突。

1.11、unix系统建议使用#!/usr/bin/env python指定解释器

系统会自动指定解释器，windows系统可以忽略

2、注释

注释和文档字符串的原则上有助于对程序的理解。python 没类型信息。如果没有注释，动态语音就很难理解。

注释不宜过多也不能太少，一般有效注释量在20%左右

写好的注释有以下建议

- 注释描述必须准确、易懂、简洁、不能有二义性。
- 避免在注释和文档注释中使用缩写，如果有缩写，则需要必要的说明
- 修改代码时更新相应的注释/文档字符串，以保证注释/文档字符串与代码的一致性
- 有含义的变量如果不能自注释，则需要添加必要的注释
- 全局变量建议添加详细注释，包括对其功能、取值范围、哪些函数或过程修改它以及存取时注意事项等说明

2.1、类和接口的文档字符串写在类申明所在的下一行，缩进四个空格

类和文档的接口字符串的内容可以选择（包括但不限于）功能描述，接口清单等。

```
# 正例
class Stonebird:
    """
    功能描述：
    接口：
    """
```

2.2、功能函数的文档字符串写在函数申明的下一行，缩进四个空格

公共函数的文档字符串内容可以选择（包括但不限于）功能描述、输入参数、输出参数、返回值、调用关系、异常描述等

```
# 正例
def load_file(path):
    """
    功能描述：
    参数：
    返回值：
    异常描述：
    """
```

2.3、公共属性的注释写在属性声明的上方，与申明保持同样的缩进

`#`文字以一个空格隔开

```
# 正例
# compentsate for border
x = x + 1
```

2.4、模块文档字符串应该写文件顶部

文档字符串应该包含功能和版权说明

```
# 正例
"""
功能：
版权信息：Copyright (c) 2021 hu. All rights reserved.
"""
```

> 尾部的""" 要自成一行

```
# 反例
"""return str
xxxxxxxxx."""

# 正例
"""return str
xxxxxxxxx.
"""

"""return str xxxxxxxxx."""
```

2.5、注释必须与其描述的代码保持同样的缩进，并放在上方相邻的位置

不可以放在下方

3、命名

3.1、包、模块使用意义完整的英文描述，采用小写lower_with_under的风格命名

模块应该采用小写加下划线的方式（lower_with_under.py）命名，尽管现存的模块使用CapWords.py命名，但现在已经不推荐这么做了

```
# 正例
from sample_packege import sample_module
from sample_module import SampleClass
```

3.2、类名使用完整的英文描述，采用大驼峰的命名风格

```
# 正例
class SampleClass(Object):
    pass
```

3.3、函数、方法、参数使用完整的英文描述，使用lower_with_under风格

```
# 正例
class SampleClass(Object):
    def sample_method(self,args:str)
        pass
```

3.4、变量使用lower_with_under，常量使用CAPS_WITH_UNDER风格

```
# 正例
sample_global_veriable = 0
M_GLOBAL_CONSTANS = 0
```

3.5、类或对象的私有成员一般采用单下划线_开始，双下划线开始__属于保护类型成员

python 没有严格的私有权限控制，业界规定使用_开头，按时此类成员仅供内部使用。

__开头会被解释器自动改名，其作用是防止在类基础场景中出现名字冲突

```
# 正例
class MyClass(object):

    def my_func(self):
        self._member = 1 # 单下划线开头，按时成员仅供内部操作使用，外部不应该访问


    def _my_private_fun(self): # 单下划线开头，按时成员仅供内部操作使用，外部不应该访问
        pass


class Mapping(object):
    def __init__(self):
        self.__update = "" # 双下划线开头，会被解释器改名为_Mapping_update,外部修改名称后仍可访问
```

3.6、变量要有明确的定义，使用完整的单词或者大家可以理解的缩写

- 命名中若使用了特殊的约定或者缩写，建议解释说明
- 对于变量命名（循环变量除外），不允许取单个字符
- 对允许使用单个字符命名的场景，不要事宜l 和o作为变量名称。这些对于数据1和0很难辨认

4、编码

4.1、与None比较要使用is或者is not，不要使用等号

is 判断两个对象的id是否相等，”==“ 判断两个对象值是否相等。

4.2、当模块有不愿对外暴露的成员时，定义`__all__`列表，将允许外部访问的变量、函数和类的名字放进去

在模块定义了`__all__`之后，外部导入只是把`__all__`定义的内容导进去

```
# 正例
__all__ = ["sample_func"]
```

4.3、避免只是使用dict[key]方式从字典中获取value

当key不存在时，dick[key]获取value方式会报KeyError，应该使用更安全dict.get(key)方式获取value

dict.get(key) 的方式key不存在时返回None

4.4、对序列使用切片操作时，不建议使用负步进值进行切片

python提供`iter[start:end:stride]`形式的写法，stride为负会使代码难以理解

```
# 反例
a = [1,2,3,4,5,6,7,8]
a[2::2]
[3, 5, 7]
a[-2::-2]
[7, 5, 3, 1]
a[-2:2:-2]
[7, 5]
a[2:2:-2]
[]
```

4.5、传递实例参数后，内部应该使用isinstance进行检查而不是type

函数内对参数进行检查应该使用isinstance函数，is not None、len(para) != 0 和其他方式都是不安全的。

```
# 正例
if not isinstace(sample,list):
    raise Exception("type err")
```

4.6、使用推导式代替重复的逻辑操作构造序列。但推导式必须考虑可读性，不在一个推导式中使用三个或以上的for语句

推导式是一种精炼的序列生成写法，可以在推导式中完成简单逻辑。

```
# 正例
num_list = [i for i in range(100) if i % 2 == 1]

# 反例
# 使用了三个for
length = [1, 2]
width = [3, 4]
heights = [5, 6]
cubs = [(x, y, z) for x in length for y in width for z in heights]
```

4.7、功能代码应该封装在函数和类中

所有顶级代码在模块导入时都会被执行，容易产生调用函数、创建对象的错误操作，所以代码应该封装在函数或者类中。

保持代码层次感

4.8、需要精确数字计算的场景，应使用decimal模块，且不要用浮点数构造

python中浮点数无法被精确表示，多次计算后可能出现尾差

4.9、避免在无法变量或者无关的概念之间重用名字。避免因重名而导致意外赋值和引用错误

```
# 反例
for x in l :
    for x in x: # 不符合，x重名
```

4.10、类的方法不需要访问实例时，根据场景选择@staticmethod或者@classmethod进行装饰

类里面定义不含self的函数，需要加上@staticmethod装饰。外部可以直接调用

```
# 正例
class MyClass:
    @staticmethod
    def my_func():
        pass
```

方法不需要访问实例成员，但需要访问派生类成员，这时应该用@classmethod

```
# 正例
class Spam:
    @classmethod
    def count(cls):
        cls.num -= 1
```

4.11、当多个python源码文件分不同子目录时，用package形式管理各个目录下的模块

通过让字目录拥有`__init__.py`文件，可以让python代码在import和from语句中，将子目录作为包名，通过分层管理各模块，让模块间的关系更清楚

4.12、避免在代码中修改sys.path列表

sys.path是python 解释器在执行import和from语句时参考的模块搜索路径，修改了sys.path可能会导致模块收搜出错

4.13、尽量使用for x in iter的方式循环处理数据，不使用for x in range(x)的方式

for i in range(x)的方式获取下标的方式是C语音编程习惯。有很多缺点，如果在循环体内修改下标，容易越界、可读性差

```
# 反例
for i in range(my_list):
    print(my_list[i])

# 正例
for x in my_list:
    print(x)
```

4.14、避免变量在其生命周期内类型发生变化

变量类型发生变化，可能导致运行时错误，造成代码复杂度提升、难以调试和维护

```
# 反例
items = 'a,b,c,d'
items = items.split(',') # 字符串变更为列表

# 正例
items = 'a,b,c,d'
items_list = items.split(',') # 字符串变更为列表
```

5、异常处理

5.1、使用try...except..结构对代码作保护时，需要在异常后使用finally结构保证操作对象的释放

```
# 正例
try:
    f = open(r'c.txt', 'a')
    strs = ["aa\n", "bb\n", "cc\n"]
    f.writelines(strs)
except BaseException as e:
    print(e)
finally:
    f.close()
    print("close file handle")
```

5.2、不要使用except：语句来捕获所有异常

对应的异常用对应的异常来捕获

5.3、不在except分支例的raise都必须带异常

`raise`关键字单独使用只能出现在`try-except`语句中，重新抛出`except`抓住的异常。

```
# 正确示例1：raise一个Exception或自定义的Exception
>>> a = 1
>>> if a==1:
...     raise Exception
...
```

5.4、尽量用异常来表示特殊情况，而不要返回None

当我们在一个工具方法时，通常会返回`None`来表明特殊的意义，比如一个数除以另外一个数，如果被除数为零，那么就返回`None`来表明是没有结果的。

```
def divide(a, b):
    try:
        return a/b
    except ZeroDivisionError:
        return None
        result = divide(x, y)
    if result is None:
        print('Invalid inputs')
```

需要记住的：
(1)方法使用`None`作为特殊含义做为返回值是非常糟糕的编码方式，因为`None`和其它的返回值必须要添加额外的检查代码。
(2)触发异常来标示特殊情况，调用者会在捕获异常来处理。

5.5、不在finally中使用return或者break

避免`finally`中可能发生的陷阱，不要在`finally`中使用`return`或者`break`语句

5.6、禁止使用except X, x语法

禁止使用`except X, x`语法，应当使用`except X as x`

`except X, x`语法只在`2.X`版本支持，`3.X`版本不支持，有兼容性问题。而且，`except X, x`写法容易和多异常捕获的元组（`tuple`）表达式混淆。因此应该统一用`except X as x`方式。

5.7、assert语句通常只在测试代码中使用，禁止在生产版本中包含assert功能

assert语句用来声明某个条件是真的。

6、并发与并行

6.1、多线程适用于阻塞式IO场景，不适用于并行计算场景

`CPython`执行`Python`代码分为2个步骤：首先，将文本源码解释编译为字节码，然后再用一个解释器去解释运行字节码。字节码解释器是有状态的，需要维护该状态的一致性，因此使用了`GIL`（`Global Interpreter Lock`，全局解释器锁）。

```
# -*- coding:utf-8 -*-
from time import time
from threading import Thread


def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i


class FactorizeThread(Thread):
    def __init__(self, number):
        Thread.__init__(self)
        self.number = number
```

```
    def run(self):
        self.factors = list(factorize(self.number))


def test(numbers):
    start = time()
    for number in numbers:
        list(factorize(number))
        end = time()
        print('Took %.3f seconds' % (end - start))


def test_thread(numbers):
    start = time()
    threads = []
    for number in numbers:
        thread = FactorizeThread(number)
        thread.start()
        threads.append(thread)
        for t in threads:
            t.join()
            end = time()
            print('Mutilthread Took %.3f seconds' % (end - start))


if __name__ == "__main__":
    numbers = [2139079, 1214759, 1516637, 1852285]
    test(numbers)
    test_thread(numbers)

# 代码输出：
# Took 0.319 seconds
# Mutilthread Took 0.539 seconds
```

6.2、建议使用Queue来协调各线程之间的工作

```
from Queue import Queue
from threading import Thread


def download(item):
    print('download item')
    return item


def resize(item):
    print('resize item')
    return item


def upload(item):
    print('upload item')
    return item


class ClosableQueue(Queue):
```

```
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)

        def __iter__(self):
            while True:
                item = self.get()
                try:
                    if item is self.SENTINEL:
                        return  # Cause the thread to exit
                    yield item
                finally:
                    self.task_done()


class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super(StoppableWorker, self).__init__()
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.func = func

    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)


if __name__ == '__main__':
    download_queue = ClosableQueue()
    resize_queue = ClosableQueue()
    upload_queue = ClosableQueue()
    done_queue = ClosableQueue()
    threads = [
        StoppableWorker(download, download_queue, resize_queue),
        StoppableWorker(resize, resize_queue, upload_queue),
        StoppableWorker(upload, upload_queue, done_queue),
    ]
    for thread in threads:
        thread.start()
    for _ in range(1000):
        download_queue.put(object())
        download_queue.close()
        download_queue.join()
        resize_queue.close()
        resize_queue.join()
        upload_queue.close()
        upload_queue.join()
        print('%s items finished' % done_queue.qsize())
```

6.3、协程

建议使用协程来处理并发场景

Python程序员可以使用线程来运行多个函数，使这些函数看上去好像是在统一时间得到执行，然而，线程有其显著的缺点：

- 多线程的运行协调起来比单线程过程式困难，需要依赖Lock来保证自己的多线程逻辑正确。
- 每个在执行的线程，大约需要8MB内存，在线程处理逻辑较少而数量较多的工程模型中开销较大。
- 线程启动、切换线程上下文的开销较大。

6.4、并行

建议使用concurrent.futures实现并行计算
Python程序可以将独立的计算任务分配到多个CPU核上运行，提升并行计算的能力。Python的GIL使得无法使用线程实现真正的并行计算。Python程序的真正并行计算建议采用子进程的方式来实现，具体实现如下：

- 对于并行计算的任务与主进程之间传递的数据比较少，且任务之间不需要共享状态和变量时，采用concurrent.furures的ProcessPoolExecutor类的简单实用方式即可实现并行计算；
- 对于并行计算的场景不满足1）的状态时，可以采用multiprocessing模块提供的共享内存、进程锁、队列、代理等高级功能实现并行计算。因为使用复杂，所以如果不是特性场景，不建议使用这种方式

使用concurrent.furures的ProcessPoolExecutor类实现并行计算的示例代码如下：

```
def calc_process():
    start = time.time()
    pool = ProcessPoolExecutor(max_workers=4)
    results = list(pool.map(gcd, numbers))
    end = time.time()
    print('process calc, Took %.3f seconds' % (end - start))
    print(results)
```

7、性能

7.1、List容量初始化

在list成员个数可以预知的情况下，创建list时需预留空间正好容纳所有成员的空间

说明：与Java、C++等语言的list一样，Python语言的list在append()成员时，如果没有多余的空间容纳新的成员，就会分配一块更大的内存，并将原来内存里的成员拷贝到新的内存上，并将最新append()的成员也拷贝到此新内存空间中，然后释放老的内存空间。如果append()调用次数很大，则如上过程会频繁发生，因而会造成灾难性性能下降，而不仅仅是一点下降。

```
# 错误示例：
members = []
for i in range(1, 1000000):
    members.append(i)
    len(members)

# 正确示例：
members = [None] * 1000000
for i in range(1, 1000000):
    members[i] = i
    len(members)
```

7.2、元素个数确定时推荐使用Tuple

在成员个数及内容皆不变的场景下尽量使用tuple替代list

说明：list是动态array，而tuple是静态array（其成员个数以及内容皆不可变）。因此，list需要更多的内存来跟踪其成员的状态。
此外，对于成员个数小于等于20的tuple，Python会对其进行缓存，即当此tuple不再使用时，Python并不会立即将其占用的内存返还给操作系统，而是保留以备后用。

```
# 错误示例：
myenum = [1, 2, 3, 4, 5]
# 正确示例：
# 如果恰好被缓存过，则初始化速度会为错误示例中的5倍以上。
myenum = (1, 2, 3, 4, 5)
```

7.3、推荐使用局部变量引用频繁使用的外界对象

对于频繁使用的外界对象，尽量使用局部变量来引用之

```
# 错误示例：
from math import tan
def afunc():
    for x in range(100000):
        return tan(x)
# 在这个例子中，Python会先到 globals() 的字典中查找tan()函数（其已经被from math import tan语句加载到了globals()中）；
# 然后在当前函数的locals()中查找x。这里存在着2次查找，比前一个例子少了一次查找，但是还不是最优解。

# 正确示例：
import math
def afunc(tan=math.tan):
    for x in range(100000):
        return tan(x)
```

7.4、尽量使用generator comprehension代替listcomprehension

list comprehension可以用来代替lambda表达式的map、reduce语法，从已有的list中，生成新的数据。而generator comprehension无需定义一个包含yield语句的函数，就可以生成一个generator。 二者一个生成list，另外一个生成generator，在内存的占用上，相差悬殊；在生成速度上，相差无几。

```
# 错误示例：
even_cnt = len([x for x in range(10) if x % 2 == 0])
# 正确示例：
even_cnt = sum(1 for x in range(10) if x % 2 == 0)
```

7.5、使用字符串格式化方式代替"+"和"+="操作符

使用`format`方法、"`%`"操作符和`join`方法代替"`+`"和"`+=`"操作符来完成字符串格式化

即使参数都是字符串，也可以使用`format`方法或`%`运算符来格式化字符串。一般性能要求的场景可以使用`+`或`+=`运算符，但需要避免使用`+`和`+=`运算符在循环中累积字符串。由于字符串是不可变的，因此会产生不必要的临时对象并导致二次而非线性运行时间。

```
# 推荐做法：
```

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

# 不推荐做法：
x = imperative + ', ' + expletive + '!'
x = 'name: ' + name + '; score: ' + str(n)
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>'
```

8、编程实践

8.1、可变参数默认值设为None

函数参数中的可变参数不要使用默认值，在定义时使用None
说明：参数的默认值会在方法定义被执行时就已经设定了，这就意味着默认值只会被设定一次，当函数定义后，每次被调用时都会有"预计算"的过程。当参数的默认值是一个可变的对象时，就显得尤为重要，例如参数值是一个list或dict，如果方法体修改这个值(例如往list里追加数据)，那么这个修改就会影响到下一次调用这个方法，这显然不是一种好的方式。应对种情况的方式是将参数的默认值设定为None。

```
# 反例
>>> def foo(bar=[]): # bar is optional and defaults to [] if not specified
...     bar.append("baz") # but this line could be problematic, as we'll see...
...     return bar
>>> foo()
["baz"]
>>> foo()
["baz", "baz"]
>>> foo()
["baz", "baz", "baz"]

# 正确示例：None是不错的选择
>>> def foo(bar=None):
...     if bar is None: # or if not bar:
...         bar = []
...     bar.append("baz")
...     return bar
...
>>> foo()
["baz"]
>>> foo()
["baz"]
>>> foo()
["baz"]
```

8.2、对子类继承的变量要做显式定义和赋初值

在`Python`中，类变量都是作为字典进行内部处理的，并且遵循方法解析顺序（`MRO`）。子类没有定义的属性会引用基类的属性值，如果基类的属性值发生变化，对应的子类引用的基类的属性的值也相应发生了变化。

```
# 正确示例：
# 如果希望类`C`中的`x`不引用自`A`类，可以在`C`类中重新定义属性`X`，
# 这样`C`类的就不会引用`A`类的属性`x`了，值的变化就不会相互影响。
class B(A):
    x = 2
class C(A):
    x = -1
>>> print(A.x, B.x, C.x)
1 2 -1
>>> A.x = 3
>>> print(A.x, B.x, C.x)
3 2 -1
```

8.3、禁使用注释行等形式仅使功能失效

python的注释包含：单行注释、多行注释、代码间注释、doc string等。除了doc string是使用"""""""括起来的多行注释，常用来描述类或者函数的用法、功能、参数、返回等信息外，其余形式注释都是使用#符号开头用来注释掉#后面的内容。基于python语言运行时编译的特殊性，如果在提供代码的时候提供的是py文件，即便是某些函数和方法在代码中进行了注释，别有用心的人依然可以通过修改注释来使某些功能启用；尤其是某些接口函数，如果不在代码中进行彻底删除，很可能在不知情的情况下就被启用了某些本应被屏蔽的功能。因此根据红线要求，在python中不使用的功能、模块、函数、变量等一定要在代码中彻底删除，不给安全留下隐患。即便是不提供源码py文件，提供编译过的pyc、pyo文件，别有用心的人可以通过反编译来获取源代码，可能会造成不可预测的结果。

> 未使用的代码直接删除掉

8.4、慎用copy和 deepcopy

在python中，对象赋值实际上是对象的引用。当创建一个对象，然后把它赋给另一个变量的时候，python并没有拷贝这个对象，而只是拷贝了这个对象的引用。如果需要拷贝对象，需要使用标准库中的copy模块。copy模块提供copy和deepcopy两个方法：

> copy浅拷贝：拷贝一个对象，但是对象的属性还是引用原来的。对于可变类型，比如列表和字典，只是复制其引用。基于引用所作的改变会影响到被引用对象。
> deepcopy深拷贝：创建一个新的容器对象，包含原有对象元素（引用）全新拷贝的引用。外围和内部元素都拷贝对象本身，而不是引用。

```
# 示例：
>>> import copy
>>> a = [1, 2, ['x', 'y']]
>>> b = a
>>> c = copy.copy(a)
>>> d = copy.deepcopy(a)
>>> a.append(3)
>>> a[2].append('z')
>>> a.append(['x', 'y'])
>>> print(a)
[1, 2, ['x', 'y', 'z'], 3, ['x', 'y']]
>>> print(b)
[1, 2, ['x', 'y', 'z'], 3, ['x', 'y']]
>>> print(c)
```

```
[1, 2, ['x', 'y', 'z']]
>>> print(d)
[1, 2, ['x', 'y']]
```

8.5、使用os.path库中的方法代替字符串拼接来完成文件系统路径的操作

`os.path` 库实现了一系列文件系统路径操作方法，这些方法相比单纯的路径字符串拼接来说更为安全，而且为用户屏蔽了不同操作系统之间的差异。

pathlib是第二选择

```
# 错误示例：如下路径字符串的拼接在Windows操作系统无法使用
path = os.getcwd() + '/myDirectory'

# 正确示例：
path = os.path.join(os.getcwd(),'/myDirectory')
# 正确示例：
path = pathlib.Path(os.getcwd(), 'myDirectory')
path = pathlib.Path().resolve() / 'myDirectory'
```

8.6、使用subprocess模块代替os.system模块来执行shell命令

说明：subprocess模块可以生成新进程，连接到它们的input/output/error管道，并获取它们的返回代码。该模块旨在替换os.system等旧模块，相比os.system模块来说更为灵活。

```
# 推荐做法：
>>> subprocess.run(["ls", "-l"]) # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)
>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

8.7、建议使用with语句操作文件

Python 对一些内建对象进行改进，加入了对上下文管理器的支持，可以用于 with 语句中。使用 with 语句可以自动关闭文件，减少文件读取操作错误的可能性，在代码量和健壮性上更优。注意 with 语句要求其操作的类型实现 **enter**() 和 **exit**() 方法，需确认实现后再使用。

```
# 推荐做法：
with open(r'somefileName') as somefile:
    for line in somefile:
        print(line)
        # ...more code
# 此使用with语句的代码等同于以下使用try...finally...结构的代码。

somefile = open(r'somefileName')
try:
    for line in somefile:
```

```
        print(line)
        # ...more code
finally:
    somefile.close()
```

9、安全编码规范

9.1、异常行为

9.1.1、禁止抑制或者忽略已检查异常

编码人员常常会通过一个空的或者无意义的`except`块来抑制捕获的已检查异常。每一个`except`块都应该确保程序只会在继续有效的情况下才会继续运行下去。因此，except块必须要么从异常情况中恢复，要么重新抛出适合当前except块上下文的另一个异常以允许最邻近的外层try-except语句块来进行恢复工作。异常会打断应用原本预期的控制流程。例如，try块中位于异常发生点之后的任何表达式和语句都不会被执行。因此，异常必须被妥当处理。许多抑制异常的理由都是不合理的。例如，当对客户端从潜在问题恢复过来不抱期望时，一种好的做法是让异常被广播出来，而不是去捕获和抑制这个异常。

9.1.2、禁止在异常中泄露敏感信息

敏感数据的范围应该基于应用场景以及产品威胁分析的结果来确定。典型的敏感数据包括口令、银行账号、个人信息、通讯记录、密钥等。如果在传递异常的时候未对其中的敏感信息进行过滤常常会导致信息泄露，而这可能帮助攻击者尝试发起进一步的攻击。

9.1.3、方法发生异常时要恢复到之前的对象状态

当发生异常的时候，对象一般需要（关键的安全对象则必须）维持其状态的一致性。常用的可用来维持对象状态一致性的手段包括：输入校验（如校验方法的调用参数）调整逻辑顺序，使可能发生异常的代码在对象被修改之前执行当业务操作失败时，进行回滚对一个临时的副本对象进行所需的操作，直到成功完成这些操作后，才把更新提交到原始的对象避免去修改对象状态。

```

```

9.2、运行环境

9.2.1、生产代码不能包含任何调试入口点

由于调试或者测试目的，开发者经常在代码中添加特定的调测代码，这些代码并没有打算与应用一起交付或者部署。当这类的调测代码不小心被留在了应用中，这个应用对某些特殊交互就是开放的。这些后门入口点可以导致安全风险。

```
// 错误代码
def wrong(flag):
    if not flag:
        import pdb
        pdb.set_trace()
        print('.....')
        ...
# end def
```

9.2.2、使用标准的API替代操作系统的系统命令

如果可以使用标准的`API`替代运行系统命令来完成任务，则应该使用标准的`API`。
当前与`Python`相关的漏洞中命令注入占比非常大，所以在此呼吁不要直接使用系统命令，不建议使用`os.system`、`subprocess`等模块去运行系统命令，如遍历目录/文件/创建文件夹等操作。

9.2.3、 禁止从第3方源下载并使用软件包

Python之所以如此流行的原因之一是其生态做的好， 其import非常灵活，方便使用的同时也存在着安全漏洞。目前Python依赖项中有5000多个已知的安全漏洞，这些都可能导致您自己的代码中出现严重的安全漏洞。别有用心的hacker已经在篡改了大量的软件包并发布的互联网上，一旦使用就会导致不可挽回的损失。

9.3、其他

9.3.1、禁止在日志中保存口令、密钥等敏感信息

在日志中不能输出口令、密钥和其他敏感信息，口令包括明文口令和密文口令。对于敏感信息建议采取以下方法：不在日志中打印敏感信息。

9.3.2、禁止将敏感信息硬编码在程序中

如果将敏感信息（包括口令和加密密钥）硬编码在程序中，可能会将敏感信息暴露给攻击者。无需反编译`pyc`文件，任何能够访问到`pyc`文件的人都可以获取这些敏感信息。因此，不能将敏感信息硬编码在程序中。

9.3.3、使用安全随机数

Python产生随机数的功能在random模块中实现，实现了各种分布的伪随机数生成器。产生的随机数可以是均匀分布，高斯分布，对数正态分布，负指数分布以及alpha，beta分布，但是这些随机数都是伪随机数，不能应用于安全加密目的的应用中。如果你需要一个真正的密码安全随机数，请使用/dev/random生成安全随机数；另外在python 3.6版本官方引入了一个secrets模块用于生成安全随机数。

```
# 【错误示例1】：
import random

sr = random.randint(0, 100)

# 【错误示例2】：
import random

foo = random.SystemRandom()  # 伪随机数
print(foo.random())
print(foo.randint(0, 10))

# random 模块还提供 SystemRandom 类，它使用系统函数 os.urandom() 从操作系统提供的源生成随机数。
# 注意： os.urandom 在linux系统环境中生成的随机数不安全，不符合我司标准。
# 在Python 3.6及更高版本中添加了secrets模块，可以使用secrets模块来实现：生成安全随机数、密码及一次性密码、随机token及会话密钥等功能。

# 【正确示例1】：linux环境下推荐使用此方法
import platform
```

```
randLength = 16  # 长度请参见密码算法规范，不同场景要求长度不一样
if platform.system() == 'Linux':
    with open("/dev/random", 'rb') as file:
        sr = file.read(randLength)
        print(sr)

# 【正确示例2】：windows环境下推荐
import platform
import os

randLength = 16  # 长度请参见密码算法规范，不同场景要求长度不一样
if platform.system() == 'Windows':
    _randLst = list(os.urandom(randLength))
    print(_randLst)
    _randBytes = os.urandom(randLength)
    print(_randBytes)

# 【正确示例3】：
# 适用于python3.6之后的版本
import secrets

# 生成随机整数
number = secrets.randbelow(10)
print("Secure random number is ", number)
# Secure random number is 0
secretsGenerator = secrets.SystemRandom()
randomNumber = secretsGenerator.randint(0, 50)
print("Secure random number is ", randomNumber)
# Secure random number is 26
# 指定范围并设置步长
randomNumber = secretsGenerator.randrange(5, 50, 5)
print("Secure random number within range is ", randomNumber)
# Secure random number within range is 15
# 从指定的数据集中选择
number_list = [6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
secure_choice = secretsGenerator.choice(number_list)
print("Secure random choice using secrets is ", secure_choice)
# Secure random choice using secrets is 30
secure_sample = secretsGenerator.sample(number_list, 3)
print("Secure random sample using secrets is ", secure_sample)
# Secure random sample using secrets is [54, 6, 42]
# 随机安全浮点数
secure_float = secretsGenerator.uniform(2.5, 25.5)
print("Secure random float number using secrets is ", secure_float)
# Secure random float number using secrets is 9.445927455984885
```

9.3.4、代码发布前务必书写开发者信息及包含开发者信息的注释内容

`RPA`层面为了防止人员流动引起的后期维护困难，所有`python`脚本代码文件开头书写开发者信息的注释内容。

9.3.5、保存外来不可信数据前要先转义

此规则来源于CSV命令注入漏洞案例。

CSV命令注入漏洞：CSV的命令注入漏洞是当用户打开存在特殊字符构造成表达式的CSV格式或Excel文档时，宿主程序（微软的Excel软件）会解析并执行该表达式，从而运行攻击者的指令，达到攻击本地计算机系统的目的。这种攻击多发生于web系统的导出日志功能上，攻击形式多样，下面是windows系统上启动了包含特殊字符串的csv文件：=cmd|'/c calc'Hi,2012lib
查看日志文档的的管理员用户（因事先了解这仅是自己的日志文件，一般会放松警惕而忽略系统的安全警示）打开该文件时就会发现系统的计算器也被执行了。

所以正确的建议就是谨慎处理用户的数据，参考前面的规则，引入黑白名单机制，或者对特殊字符进行转义，此处CSV漏洞属于特别的场景，可以对=做处理，比如在=号前面插入字符\t，如让微软的Excel程序正确解释即可。提示，包含以上字符串的csv文件被Symantec杀毒软件报毒为Trojan.Slakexec!gen4。