# 1.命名

1. 小写下划线命名

包(Package) 模块(Moudle) 函数(Function) 方法(Method)

函数参数 变量

1. 大写下划线

常亮

1. 大驼峰

类(class) 异常

1. 单下划线开头

类 对象 的私有成员

1. 双下划线开头

被继承的基类成员，防止与派生类重名

“\_\_”开头的成员会被解释器自动改名为，加上类名作为前缀

# 2.与None比较

与None比较用”is ” or “is not ”,不要使用等号

“is” 判断的 是否为同一个对象（id）

“==”判断对象的值是否相等

# 3.\_\_all\_\_使用

在模块中定义\_\_all\_\_之后，从外部from module import \* 只会

Import \_\_all\_\_ 里定义的内容

class module(object):

\_\_all\_\_=[“xxx\_function”]

# 4.dict使用

避免直接使用 dict[key] 的方式从字典中取值，如果不存在，会产生keyerror

应用dick.get(key)来取值

# 5.切片

不建议负步进值切片

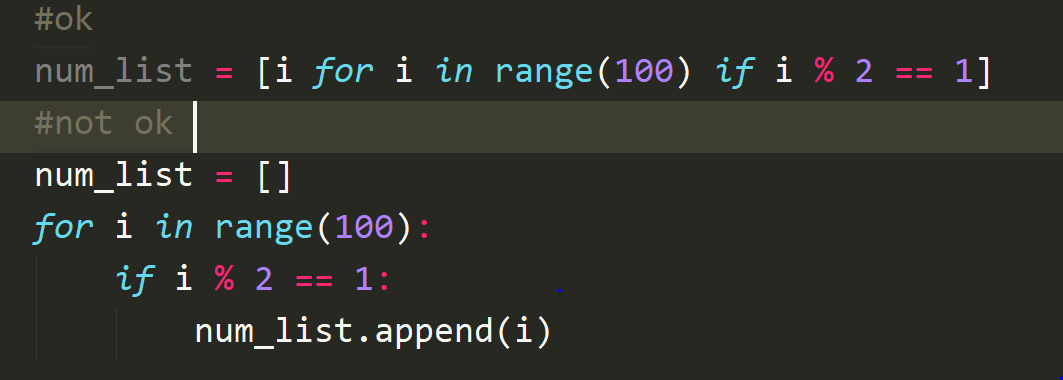
# 6.实例类型检查

传递实例类型参数后，函数内应用isinstance 进行参数检查

推导式

尽量使用推导式 代替重复的逻辑操作构造序列，但推导式必须考虑可读性，不推荐使用

两个以上的表达是的列表推导式



# 6.精确数值计算

精确的数值计算场景用decimal 模块

Ps: 不要用浮点数构造decimal，浮点数本身就I不准确

from decimal import decimal

# 7.避免在代码中修改sys.path

# 8.LEGB

Python 解析引用名字 local—enclosed—global—builtin

变量不要同名

建议使用for x in iterable

# 9.@staticmethod和 @classmethod 使用

类的方法不需要访问实例时，根据具体场景选择@staticmethod 和@classmethod进行

修饰

说明：

一般的类方法要接收一个self参数表示此类实例，但是有点方法不需要访问实例，这时分两种

情况

a. 方法不需要访问任何成员或只需要显示访问这个类自己的成员，这样的方法不需要额外的参数，应该用@staticmethod

在python3.x 中，允许直接定义不含有self 参数的方法，并且允许不通过实例调用，但一旦实例调用这个方法就会因为参数不匹配而出错，加@staticmethod 可以让python解释器明确此方法不需要self参数提前拦截问题

b.方法不需要访问实例的成员，但需要访问基类或派生的成员，这时用@classmethod装饰，装饰后的 方法，其第一个参数不再传入实例，而是传入调用者的最底层类：

但是在使用@classmethod时，需要注意，由于继承场景下传入的第一参数并不一定是这个类本身，因此并非所有访问类成员的场景都应该用@classmethod。

# 10.异常捕获使用

* Try ….except…结构对代码的保护时，需要在finally结构中保护操作对象的释放
* 不要使用except 捕获所有的异常，except捕获需要处理的异常，其他预料外的异常让上层函数捕获
* 不在except里的raise都必须带异常

a =1

if a == 1 :

raise 错误 raise exception 正确

* 尽量用异常来表示特殊的情况，而不要反回None

def div(a,b):

try:

return a/b

except xxxx\_error as e :

#return None 错误

raise vlaueError(“xxxxx”) from e 正确

* finally 里不能使用return break

# 11.数组使用

* list 成员个数确定时，创建list 时需预留空间正好能够容纳所有成员

#members = [] 错误

members = [None] \* 1000 正确 动态数组的问题

for i in range(1,1000):

members.append(i)

* 在成员个数及内容都不变的情况下，尽量使用tuple 替代list

Ps：1.list 是动态array 而tuple 是静态array（其成员个数和内容都不可变）

2.list 需要更多的内存来跟踪其成员状态

3.对于成员个数小于等于 20的tuple ，python 会对其进行缓存，即当tuple不在使用时，pyhton 并不会立即释放占用内存

my\_enum = [1,2,3,4,5]

my\_enum = (1,2,3,4,5) 如果缓存过，初始化速度是list的5倍以上

* 用生成器来替代list推导器

# 12.字符串格式化

用format 方法, % 操作符，join 代替“+”，“+=” 操作符来完成字符串的格式化

一般性能要求 的场景可以使用 “+””+=”运算符，但需要避免 “+” ”+=”在循环中累积字符串，因为字符串是不可变得，因此会产生很多不必要的临时对象，导致性能问题

# 13.函数中可变参数

函数参数中的可变参数不要使用默认值，在定义时使用None

错误示例

def foo(bar=[]):

bar.append(“a”)

return bar

正确示例

def foo(bar=None):

if bar is None:

bar = []

bar.append(“a”)

return bar

# 14.文件系统路径拼接

os.path 库中的方法代替字符串拼接来完成文件系统的路径操作

相对于单纯的路径字符串拼接来说更安全

错误 path = os.getcwd() + ‘/mydir’

正确 path = os.path.join(os.getcwd(),’mydir’)

# 15.Python执行shell命令

使用subprocess 模块来代替os.system模块来执行shell 命令

说明： subprocess 模块 可以生成新进程，连接到他们的input output error 管道，并获取他们的返回码，该模块为了代替os.system等相关模块，相较os.system而言更灵活

subprocess.run[“ls”,”-l”]

# 16.文件操作用with语句

# 17.并发和并行

* 多线程适用于阻塞IO场景，不适用于并行计算场景
* 使用queue来协调各线程之间的工作
* 用协程来处理并发场景

理由对比线程

1 . 多线程协调比单线程困难 要加锁

2．每个执行线程大约需要8mb的内存

3.线程启动，切换线程的上下文开销大（相对于协程）

协程：

1.使python程序看起来好像同时运行多个函数，运行逻辑调度由开发者自己决定

2.协程的实现方式，实际上是对生成器的一种扩展

3.启动生成器协程需要的开销与调用函数相仿

活跃状态的协程 占用不到1kb的内存

* concurrent.future 实现并行

python 程序可以将独立的计算任务分配到多个cpu核上运行，提升并行计算能力，

python GIL 让线程无法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实现，

具体实现如下：

1.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.futures中的processPoolExecutor类的方法来实现并行计算

2.对于并行计算不满足1的情况，可以使用multiprocessing模块的共享内存,进程锁,代理等高级功能实现并行计算。比较复杂，不是特定场景不建议使用