Pythonic

版本查看的方法

```
!python  --version
```

```
Python 3.6.9
```

```
import  sys
print(sys.version_info)
print(sys.version)
```

```
sys.version_info(major=3, minor=6, micro=9, releaselevel='final', serial=0)
3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0]
```

遵循PEP8风格指南

- 采用一致的风格来编写大妈，可以令后续的修改工作变得更为容易!

了解bytes、str与Unicode的区别

- Unicide字符利用encode、decode方法完成二进制转换
- 一般用UFT-8 统一编码形式

```
str1  =  "abc"
test  =  str1.encode()
test.decode()
```

```
'abc'
```

1.接受str或bytes，总是返回str的方法

2.接受str或bytes，总是返回bytes的方法

```
def  to_str(bytes_or_str):
    if  isinstance(bytes_or_str,bytes):
        value  =  bytes_or_str.decode('utf-8')
    else:
        value  =  bytes_or_str
    return  value

def  to_bytes(bytes_or_str):
    if  isinstance(bytes_or_str,str):
        value  =  bytes_or_str.encode('utf-8')
    else:
        value  =  bytes_or_str
    return  value
```

```
return value
```

- 可以利用+操作符吧str与unicode连接起来
- 可以利用等价于不等价操作符，对str和Unicode实例进行比较
- 在格式字符串中，可以利用“%s”等形式来代表unicode实例
- python3中 利用open函数获取了文件句柄，默认会采用UTF-8编码格式来操作文件
- 从文件中读取二进制，或向其中写入二进制数据时，总应该1以‘rb’或‘wb’等二进制模式来开启

用辅助函数来取代复杂的表达式

- 请把复杂的表达式移入辅助函数中，如果要反复使用相同的逻辑，那就更应该这么做

了解切割序列的办法

slice（切片）操作

- 可以对内置的list、str和bytes进行切割
- 延伸到**getitem**和**setitem**上使用

切割的基本操作 sonelist[start:end]

start所指的元素涵盖在切割后的范围内,**end**所指的元素则不包含在切割结果中

在单词切片操作内,不要同时指定start/end/stride

somelist[start:end:stride]用来实现从每n个元素中取出一个来

指定步进值**(stride)**

- 尽量使用stride为正数,且不带start或end索引的切割操作.尽量避免使用负数做stride
- 同一个切片操作上,不要同时使用start/end/stride

用列表推导来取代map和filter

利用列表推导要比使用map更清晰

```
a = [1,2,3,4,5]
squares1 = [x**2 for x in a]
squares2 = map(lambda x: x** 2,a)
```

不要使用含有两个以上表达式的列表推导

- 列表推导支持多级循环,每一级循环也支持多项条件
- 超过两个表达式的列表推导是很难理解的,应该尽量避免

用生成器表达式来改写数据量较大的列表推导

在推导过程中,对于输入序列中的每个值来说,可能要创建仅含一项元素的全新列表.数据非常多时,会

致程序崩溃

```
it = (len(x) for x in open('tmp/my_file.txt'))
```

尽量用enumerate取代range

enumerate函数可以把各种迭代器包装为生成器,以便后续产生输出值.生成器每次产生一对输出值,前后者表示从迭代器中获取到的下一个序列元素.

```
flavor_list = ['vanilla','chocolate','pecan','strawberry']
for i,flavor in enumerate(flavor_list):
    print('%d:%s'%(i+1,flavor))
```

```
1:vanilla
2:chocolate
3:pecan
4:strawberry
```

- enumerate函数提供了在遍历迭代器时可以获取每个元素索引
- 可以给enumerate提供第二个参数,以指定开始计数时所用的值

用zip函数同时遍历两个迭代器

关联表中可以用zip函数 把两个或以上的迭代器封装为生成器 从每个迭代器中获取该迭代器的下一个祖(tnple)

```
names = ['Cecilia','Lise','Marie']
letters = [len(n) for n in names]
```

普通写法

```
longest_name = None
max_letters = 0
for i in range(len(names)):
    count = letters[i]
    if count > max_letters:
        longest_name = names[i]
        max_letters = count
print(longest_name)
```

```
Cecilia
```

enumerate写法

```
for i,name in enumerate(names):
    count = letters[i]
    if count>max_letters:
```

```
        longest_name = name
        max_letters = count
print(longest_name)
```

```
Cecilia
```

zip写法

```
for name,count in zip(names,letters):
    if count>max_letters:
        longest_name = name
        max_letters = count
print(longest_name)
```

```
Cecilia
```

- zip函数可以平行地遍历多个迭代器
- python3中的zip相当于生成器,会在遍历过程中逐次产生元祖,而python2中的zip则是直接把元素
  次性的返回整份列表
- 如果zip中提供的迭代器长度不等,zip会自动提前终止
- itertools内置模块中的zip_longest函数可以平行地遍历多个迭代器,不用在乎长度

不要在for和while循环后面写else块

- 不要在循环后面使用else块,既不直观又容易误解

合理利用try/except/else/finally结构中的每个代码块

异常处理的四种不同时机:

try/except/else/finally

finally结构用于确保程序能够可靠地关闭文件句柄

```
handle = open('/tmp/random_data.txt')
try:
    data = handle.read()
finally:
    handle.close()
```

else块能清晰地描述出哪些异常会由自己的代码处理/哪些异常会传播到上一级.

```
def load_json_key(data,key):
    try:
        result_dict = json.loads(data)
    except ValueError as e:
        raise KeyError from e
    else:
        return result_dict[key]
```

顺利运行try块后,若想使某些操作能在finally块的清理代码之前执行,则可将这些操作写到else块中

函数

尽量用异常来表示特殊情况,而不要返回None

函数在遇到特殊情况时,应该抛出异常,而不要返回None

了解如何在闭包里使用外围作用域中的变量

闭包:一种定义在某个作用域中的函数,这种函数引用了那个作用域里面的变量

python的函数是一级对象(first-class object),可以直接引用函数/把函数赋给变量/把函数当成参数传
表达式及if语句判断比较等

- 在python3中,程序可以在闭包内用nonlocal语句来修饰某个名称,使该闭包能够修改外围作用域

考虑用生成器来改写直接返回列表的函数

由生成器函数所返回的那个迭代器,可以把生成器函数体中,传给yield表达式的那些值,逐次产生出来

```
def  index_words_iter(text):
    if  text:
        yield  0
    for  index,letter  in  enumerate(text):
        if  letter==  '  ':
            yield  index  +  1
```

在参数上迭代时,要多加小心

迭代器协议容器类的编写(iterator protocol)

在 'for x in foo'中,python实际上会调用iter(foo).内置的iter函数又会调用'foo.**iter**'这个特殊的方法,该方
器对象,而那个迭代器本身,则实现了next的特殊方法, for循环会在迭代器对象上面反复调用内置的ne
尽并产生StopIteration异常

```
class  ReadVisite(object):
    def  __init__(self,data_path):
        self.data_path  =  data_path

    def  __iter__(slef):
        with  open(self.data_path)  as  f:
            for  line  in  f:
                yield  int(line)

visits  =  ReadVisits(path)
percentages  =  nomalize(visits)
```

```
print(percentages)
```

- 想判断某个值是迭代器还是容器,可以拿该值为参数,两次调用iter函数,若结果相同,则是迭代器,调
  next函数,即可令迭代器前进一步.

用数量可变的位置参数减少视觉杂讯

利用*实现参数可选调用;

```
def log(message,*values):
    if not values:
        print(message)
    else:
        values_str = ','.join(str(x)for x in values)
        print('%s: %s'%(message,values_str))

log("My numbers are",1,2)
log('Hi there')
```

```
My numbers are: 1,2
Hi there
```

双击（或按回车键）即可修改

- 在def语句中使用*args,可令函数接受数量可变的位置参数
- 对生成器使用*操作符,可能导致程序耗尽内存并崩溃
- 在已经接受*args参数的函数上面继续添加位置参数,可能会产生难以排查的bug

用关键字参数来表达可选的行为

- 函数参数可以按位置或关键字来指定
- 关键字参数能够阐明每个参数的意图
- 可选的关键字参数,总是应该以关键字形式来指定,而不应该以位置参数的形式来指定

用None和文档字符串来描述具有动态默认值的参数

??

对于以动态值作为实际默认值的关键字参数来说,应该把形式上的默认值写为None,并在函数的文档字
默认值所对应的实际行为

用只能以关键字形式指定的参数来确保代码明晰

- 对于各参数之间很容易混淆的函数,可以声明只能以关键字形式指定的参数,以确保调用者必须以
  它们.

- Python3有明确的语法来定义这种只能以关键字形式指定的参数

类与继承

尽量用辅助类来维护程序的状态,而不要用字典和元组

如果保存内部状态的字典变得比较复杂,那就应该把代码拆解为多个辅助类

简单的接口应该接受函数,而不是类的实例

API执行时,通过挂钩(hook)函数,回调函数内的代码

如果要用函数来保存状态,那就应该定义新的类,并令其实现call方法,而不要定义带状态的闭包

以@classmethod形式的多态去通用地构建对象

多态:使得继承体系中的多个类都能以各自所独有的方式来实现某个方法.

MapReduce流程

Map阶段:

Step 1：读取输入文件的内容，并解析成键值对（<key, value>）的形式，输入文件中的每一行被解
value>对，每个<key, value>对调用一次map()函数。

Step 2：用户写map()函数，对输入的<key,value>对进行处理，并输出新的<key,value>对。

Step 3：对Step 2中得到的<key,value>进行分区操作。

Step 4：不同分区的数据，按照key值进行排序和分组，具有相同key值的value则放到同一个集合中

Step 5（可选）：分组后的数据进行规约。

Reduce阶段:

Step 1：对于多个map任务的输出，按照不同的分区，通过网络传输到不同的Reduce节点。

Step 2：对多个map任务的输出结果进行合并、排序，用户书写reduce函数，对输入的key、value进
的key、value输出结果。

Step 3：将reduce的输出结果保存在文件中。

- 在python程序中,每个类只能有一个构造器,也就是init方法
- 通过@classmethod机制,可以用一种与构造器相仿的方法来构造类的对象
- 通过类方法多态机制,可以构建并拼接具体的子类

用super初始化父类

内置super函数,定义了方法解析顺序(MRO method resolution order)

MRO以标准的流程来安排超类之间的初始化顺序,保证钻石顶部的公共基类的init方法只运行一次

python3在提供了不带参数的super调用方式,利用class和self来调用super

```
class  Explicit(MyBaseClass):
    def  __init__(self,value):
        super(__class__,self).__init__(value  *  2)

class  Implicit(MyBaseClass):
    def  __inti__(self,value):
        super().__init__(value  *  2)
```

多用public属性,少用private属性

python中的private属性以下划线开头:__private_field = 10

```
#prog_1
class  MyClass(object):
    def  __init__(self,value):
        self.__value  =  value
    def  get_value(self):
        return  str(self.__value)
foo  =  MyClass(5)
assert  foo.get_value()  ==  '5'

#prog_2
class  MyBaseClass(object):
    def  __init__(self,value):
        self.__value  =  value
        #  ...
class  MyClass(MyBaseClass):
    #  ...
class  MyIntergetSubclass(MyClass):
    def  get_value(self):
        return  int(self._MyClass.__value)

#引用失效
```

继承collections.abc以实现自定义的容器类型

python提供了管理数据所用的内置容器类型,list/tuple/set/dic等

元类及属性

用纯属性取代get和set方法

利用@property1装饰器优化设计

用描述符来改写需要复用的@property方法

WeakKeyDictionary可以保证描述符类不会泄漏内存

用getattr/getattribute/setattr实现按需生成的属性

子类通过 `super().__getattr__()` 来获取真正额属性值

利用内置函数hasattr函数来判断对象是否已经拥有了相关属性

通过 `__getattr__` 和 `__setattr__`,用惰性的方式来加载并保存对象的属性

如果要在 `__getattribute__` 和 `__setattr__` 方法中访问实例属性,那么应该直接通过super()来访问

用元类来验证子类

通过元类 可以再生成子类对象之前,先验证子类的定义是否合乎规范

python系统把子类的整个class语句体处理完毕后,就会调用其元类的 `__new__` 方法

用元类来注册子类

通用地基类,可以记录程序调用本类构造器时所用的参数,并将其转换为JSON字典

```
import  json
class  Serializable(object):
    def  __init__(self,*args):
        self.args  =  args
    def  seriablize(self):
        return  json.dumps({'args':self.args})
        #这个类将不可变的数据结构转换为字符串


class  Point2D(Serializable):
    def  __init__(self,x,y):
        super().__init__(x,y)
        self.x  =  x
        self.y  =  y
def  __repr__(self):
    return  'Point2D(%d,%d)'  %(self.x,self.y)

point  =  Point2D(5,3)
print(point)
print(point.seriablize())
```

```
<__main__.Point2D object at 0x7f8c77a1c390>
{"args": [5, 3]}
```

用元类来注解类的属性

元类可以再类定义后,还未使用的时候,提前修改或注解类的属性

注解某个类的属性,相当于在该属性上附加一些信息,以阐明其意图

```
class  Field(object):
    def  __init__(self,name):
        self.name=  name
        self.internal_name  =  '_'  +  self.name

    def  __get__(self,instance,instance_type):
        if  instance  is  None:return  self
        return  getattr(instance,self,instarnam_name,'')
    def  __set__(self,instance,value):
        setattr(instance,slef.internal_name,value)
```

并发及并行

并发(concurrency)

并行(parallelism)

并行和并发的关键区别在于能不能提速(sppedup)

用python语音编写并发程序时,通过系统调用/子进程和c语言扩展等机制,可以使python平行处理一些

但要使python代码并行运行,是相当困难的

用subprocess模块来管理子进程

用subprocess模块运行子进程

把子进程从父进程中剥离(decouple,解耦),父进程可以随意

可以给communicate方法传入timeout参数,以避免子进程死锁或失去响应(hanging,挂起)

可以用线程来执行阻塞式I/O,但不要用它做平行计算

- python采用GIL(global interpreter lock 全局解释器锁)机制来确保协调性
- GIL是互斥锁(mutex)
- 标准的Thread类无法使CPython解释器利用CPU的多个内核
- 尽管受制于GIL,但是Python的多线程功能依然可以模拟出同一时刻执行多项任务的效果
- 通过Python线程,可以平行地执行多个系统调用,这使得程序能够在执行阻塞式I/O操作的同时,执

在线程中使用Lock来防止数据竞争

利用互斥锁来保护Counter对象,使得多个线程同时访问value值,而不会破坏该值.

用Queue来协调各线程之间的工作

Queue类使得工作线程无需再频繁地查询输入队列的状态,get方法会持续阻塞,直到有新的数据加入.

```
from  queue  import  Queue
queue  =  Queue()

def  consumer():
    print('Consumer  waiting')
    queue.get()
    print('Consumer  done')

thread  =  Thread(target=consumer)
thread.start()
```

线程启动后,并不会立即执行完毕,卡在 `queue.get()` ,必须调用Queue实例的put方法,给队列中放入一项 `queue.get()` 防范返回

```
print  ('Producer  putting')
queue.put(object())
thread.join()
print('Producer  done')
```

考虑用协程来并发地运行多个函数

线程的三个显著缺点:

1. 多线程的代码比单线程更难以扩展维护
2. 过多的线程会占用大量内存
3. 线程启动的开销大

Python中的协程避免了上述问题

```
def  my_coroutine():
    while  True:
        received  =  yield
        print('Received:',received)

it  =  my_coroutine()
next(it)
it.send('First')
it.send('Second')
```

```
Received: First
Received: Second
```

调用send方法之前,先调用一次next函数以便生成器推进到第一条yield表达式那里

调用send方法之前,先调用一次next函数,以使生成器推进到第一条yield表达式那里.

yield操作与send操作结合起来使用

考虑用concurrent.futures来实现真正的平行计算

concurrent.futures multiprocessing multiprocessing 可以实现程序的加速

内置模块

用functools.wraps定义函数修饰器

修饰器(decorator)

```
def trace(func):
    def wrapper(*args,**kwargs):
        result = func(*args,**kwargs)
        print('%s(%r,%r) > %r'%(func.__name__,args,kwargs,result))
        return result
    return wrapper

# 利用@符号把这个修饰器套用到某个函数上面

@trace
def fibonacci(n):
    if n in (0,1):
        return n
    return (fibonacci(n-2) + fibonacci(n-1))
# 使用@符号来修饰函数,其效果就等于先以该函数为参数,调用修饰器,然后把修饰器所返回的结果,赋给同一个f
fibonacci(3)
```

```
fibonacci((1,),{}) > 1
fibonacci((0,),{}) > 0
fibonacci((1,),{}) > 1
fibonacci((2,),{}) > 1
fibonacci((3,),{}) > 2
2
```

python为修饰器提供了专门的语法,它使得程序在运行的时候,能够用一个函数来修改另一个函数

考虑以contextlib和with语句来改写可复用的try/finally代码

```
import logging
def my_function():
    logging.debug("Some debug data")
    logging.error('Error log here')
    logging.debug('More debug data')

my_function()
```

```
ERROR:root:Error log here
```

```
@contextmanager
def debug_logging(level):
    logger = logging.getLogger()
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield
    finally:
        logger.setLevel(old_level)
```

利用with语句实现句柄操作后自动关闭机制

```
with open('/tmp/my_output.txt','w') as handle:
    handle.write('This is some data!')
```