Python基础

文件操作

1.有一个jsonline格式的文件file.txt大小约为10K

```
def get_lines():
    with open('file.txt','rb') as f:
        return f.readlines()

if___name__== '_main_':
    for e in get_lines():
        process(e) # 处理每一行数据
```

现在要处理一个大小为**10G**的文件，但是内存只有**4G**，如果在只修改**get_lines** 函数而其他代码保持不变的情况下，应该如何实现？需要考虑的问题都有那些？

```
def get_lines():
    with open('file.txt','rb') as f:
        for i in f:
            yield i
```

个人认为：还是设置下每次返回的行数较好，否则读取次数太多。

```
def get_lines():
    l = []
    with open('file.txt','rb') as f:
      data = f.readlines(60000)
    l.append(data)
    yield l
```

Pandaaaa906提供的方法

```
from mmap import mmap


def get_lines(fp):
    with open(fp,"r+") as f:
        m = mmap(f.fileno(), 0)
        tmp = 0
        for i, char in enumerate(m):
            if char==b"\n":
                yield m[tmp:i+1].decode()
                tmp = i+1

if_name_=="_main__":
    for i in get_lines("fp_some_huge_file"):
        print(i)
```

要考虑的问题有：内存只有**4G**无法一次性读入**10G**文件，需要分批读入分批读入数据要记录每次读入数据的位置。分批每次读取数据的大小，太小会在读取操作花费过多时间。
https://stackoverflow.com/questions/30294146/python-fastest-way-to-process-large-file

2.补充缺失的代码

```
def print_directory_contents(sPath):
"""
这个函数接收文件夹的名称作为输入参数
返回该文件夹中文件的路径
以及其包含文件夹中文件的路径
"""
import os
for s_child in os.listdir(s_path):
    s_child_path = os.path.join(s_path, s_child)
    if os.path.isdir(s_child_path):
        print_directory_contents(s_child_path)
    else:
        print(s_child_path)
```

模块与包

3.输入日期， 判断这一天是这一年的第几天?

```
import datetime
def dayofyear():
    year = input("请输入年份: ")
    month = input("请输入月份: ")
    day = input("请输入天: ")
    date1 = datetime.date(year=int(year),month=int(month),day=int(day))
    date2 = datetime.date(year=int(year),month=1,day=1)
    return (date1-date2).days+1
```

4.打乱一个排好序的list对象alist?

```
import random
alist = [1,2,3,4,5]
random.shuffle(alist)
print(alist)
```

数据类型

5.现有字典 d= {'a':24,'g':52,'i':12,'k':33}请按value值进行排序?

```
sorted(d.items(),key=lambda x:x[1])
```

x[0]代表用key进行排序；x[1]代表用value进行排序。

6.字典推导式

```
d = {key:value for (key,value) in iterable}
```

7.请反转字符串 "aStr"?

```
print("aStr"[::-1])
```

8.将字符串 "k:1 |k1:2|k2:3|k3:4"，处理成字典 {k:1,k1:2,...}

```
str1 = "k:1|k1:2|k2:3|k3:4"
def str2dict(str1):
    dict1 = {}
    for iterms in str1.split('|'):
        key,value = iterms.split(':')
        dict1[key] = value
    return dict1
#字典推导式
d = {k:int(v) for t in str1.split("|") for k, v in (t.split(":"), )}
```

9.请按alist中元素的age由大到小排序

```
alist = [{'name':'a','age':20},{'name':'b','age':30},{'name':'c','age':25}]
def sort_by_age(list1):
    return sorted(alist,key=lambda x:x['age'],reverse=True)
```

10.下面代码的输出结果将是什么?

```
list = ['a','b','c','d','e']
print(list[10:])
```

代码将输出**[]**,不会产生**IndexError**错误，就像所期望的那样，尝试用超出成员的个数的**index**来获取某个列表的成员。例如，尝试获取**list[10]**和之后的成员，会导致**IndexError**。然而，尝试获取列表的切片，开始的**index**超过了成员个数不会产生**IndexError**，而是仅仅返回一个空列表。这成为特别让人恶心的疑难杂症，因为运行的时候没有错误产生，导致**Bug**很难被追踪到。

11.写一个列表生成式，产生一个公差为11的等差数列

```
print([x*11 for x in range(10)])
```

12.给定两个列表，怎么找出他们相同的元素和不同的元素?

```
list1 = [1,2,3]
list2 = [3,4,5]
set1 = set(list1)
set2 = set(list2)
print(set1 & set2)
print(set1 ^ set2)
```

13.请写出一段python代码实现删除list里面的重复元素?

```
l1 = ['b','c','d','c','a','a']
l2 = list(set(l1))
print(l2)
```

用**list**类的**sort**方法:

```
l1 = ['b','c','d','c','a','a']
l2 = list(set(l1))
l2.sort(key=l1.index)
print(l2)
```

也可以这样写:

```
l1 = ['b','c','d','c','a','a']
l2 = sorted(set(l1),key=l1.index)
print(l2)
```

也可以用遍历:

```
l1 = ['b','c','d','c','a','a']
l2 = []
for i in l1:
    if not i in l2:
        l2.append(i)
print(l2)
```

14.给定两个list A，B ,请用找出A，B中相同与不同的元素

```
A,B 中相同元素： print(set(A)&set(B))
A,B 中不同元素： print(set(A)^set(B))
```

企业面试题

15. python新式类和经典类的区别?

a. 在**python**里凡是继承了**object**的类，都是新式类

b. **Python3**里只有新式类

c. **Python2**里面继承**object**的是新式类，没有写父类的是经典类

d. 经典类目前在**Python**里基本没有应用

e. 保持**class**与**type**的统一对新式类的实例执行**a.class**与**type(a)**的结果是一致的，对于旧式类来说就不一样了。

f.对于多重继承的属性搜索顺序不一样新式类是采用广度优先搜索，旧式类采用深度优先搜索。

16. python中内置的数据结构有几种?

a. 整型 **int**、 长整型 **long**、浮点型 **float**、 复数 **complex**

b. 字符串 **str**、 列表 **list**、 元祖 **tuple**

c. 字典 **dict** 、 集合 **set**

d. **Python3** 中没有 **long**，只有无限精度的 **int**

17. python如何实现单例模式?请写出两种实现方式?

第一种方法:使用装饰器

```
def singleton(cls):
    instances = {}
    def wrapper(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args,**kwargs)
        return instances[cls]
    return wrapper
```

```
@singleton
class Foo(object):
    pass
foo1 = Foo()
foo2 = Foo()
print(foo1 is foo2)  # True
```

第二种方法：使用基类
New 是真正创建实例对象的方法，所以重写基类的**new** 方法，以此保证创建对象的时候只生成一个实例

```
class Singleton(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, '_instance'):
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)
        return cls._instance


class Foo(Singleton):
    pass

foo1 = Foo()
foo2 = Foo()

print(foo1 is foo2)  # True
```

第三种方法：元类，元类是用于创建类对象的类，类对象创建实例对象时一定要调用**call**方法，因此在调用**call**时候保证始终只创建一个实例即可，**type**是**python**的元类

```
class Singleton(type):
    def __call__(cls, *args, **kwargs):
        if not hasattr(cls,'_instance'):
            cls._instance = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instance


# Python2
class Foo(object):
    __metaclass__= Singleton

# Python3
class Foo(metaclass=Singleton):
    pass

foo1 = Foo()
foo2 = Foo()
print(foo1 is foo2)  # True
```

18.反转一个整数，例如-123 --> -321

```
class Solution(object):
    def reverse(self,x):
        if -10<x<10:
```

```
            return x
        str_x = str(x)
        if str_x[0] !="-":
            str_x = str_x[::-1]
            x = int(str_x)
        else:
            str_x = str_x[1:][::-1]
            x = int(str_x)
            x = -x
        return x if -2147483648<x<2147483647 else 0
if __name__== '_main__':
    s = Solution()
    reverse_int = s.reverse(-120)
    print(reverse_int)
```

19.设计实现遍历目录与子目录，抓取.pyc文件

第一种方法:

```
import os

def get_files(dir,suffix):
    res = []
    for root,dirs,files in os.walk(dir):
        for filename in files:
            name,suf = os.path.splitext(filename)
            if suf == suffix:
                res.append(os.path.join(root,filename))

    print(res)

get_files("./",'.pyc')
```

第二种方法:

```
import os

def pick(obj):
    if obj.endswith(".pyc"):
        print(obj)

def scan_path(ph):
    file_list = os.listdir(ph)
    for obj in file_list:
        if os.path.isfile(obj):
            pick(obj)
        elif os.path.isdir(obj):
            scan_path(obj)

if__name__=='__main__':
    path = input('输入目录')
    scan_path(path)
```

第三种方法

```
from glob import iglob


def func(fp, postfix):
    for i in iglob(f"{fp}/**/*{postfix}", recursive=True):
        print(i)

if__name__== "_main_":
    postfix = ".pyc"
    func("K:\Python_script", postfix)
```

20.一行代码实现1-100之和

```
count =sum(range(0,101))
print(count)
```

21.Python-遍历列表时删除元素的正确做法

遍历在新在列表操作，删除时在原来的列表操作

```
a = [1,2,3,4,5,6,7,8]
print(id(a))
print(id(a[:]))
for i in a[:]:
    if i>5:
        pass
    else:
        a.remove(i)
    print(a)
print('..................')
print(id(a))
```

```
#filter
a=[1,2,3,4,5,6,7,8]
b = filter(lambda x: x>5,a)
print(list(b))
```

列表解析

```
a=[1,2,3,4,5,6,7,8]
b = [i for i in a if i>5]
print(b)
```

倒序删除
因为列表总是‘向前移’，所以可以倒序遍历，即使后面的元素被修改了，还没有被遍历的元素和其坐标还是保持不变的

```
a=[1,2,3,4,5,6,7,8]
print(id(a))
for i in range(len(a)-1,-1,-1):
    if a[i]>5:
        pass
    else:
        a.remove(a[i])
print(id(a))
print('---------- ')
print(a)
```

22. 字符串的操作题目

全字母短句 **PANGRAM** 是包含所有英文字母的句子，比如：**A QUICK BROWN FOX JUMPS OVER THE LAZY DOG.** 定义并实现一个方法 **get_missing_letter,** 传入一个字符串采纳数，返回参数字符串变成一个 **PANGRAM** 中所缺失的字符。应该忽略传入字符串参数中的大小写，返回应该都是小写字符并按字母顺序排序（请忽略所有非 **ACSII** 字符）

下面示例是用来解释，双引号不需要考虑:

(0) 输入**: "A quick brown for jumps over the lazy dog"**

返回： **""**

(1) 输入**: "A slow yellow fox crawls under the proactive dog"**

返回**: "bjkmqz"**

(2) 输入**: "Lions, and tigers, and bears, oh my!"**

返回**: "cfjkpquvwxz"**

(3)输入**: ""**

返回：**"abcdefghijklmnopqrstuvwxyz"**

```
def get_missing_letter(a):
    s1 =set("abcdefghijklmnopqrstuvwxyz")
    s2 = set(a.lower())
    ret = "".join(sorted(s1-s2))
    return ret

print(get_missing_letter("python"))

# other ways to generate letters
# range("a", "z")
# 方法一:
import string
letters = string.ascii_lowercase
# 方法二:
letters = "".join(map(chr, range(ord('a'), ord('z') + 1)))
```

23. 可变类型和不可变类型

1,可变类型有**list,dict.**不可变类型有**string**，**number,tuple.**

2,当进行修改操作时，可变类型传递的是内存中的地址，也就是说，直接修改内存中的值，并没有开辟新的内存。

3,不可变类型被改变时，并没有改变原内存地址中的值，而是开辟一块新的内存，将原地址中的值复制过去，对这块新开辟的内存中的值进行操作。

24. is和==有什么区别?

is：比较的是两个对象的**id**值是否相等，也就是比较俩对象是否为同一个实例对象。是否指向同一个内存地址

==： 比较的两个对象的内容/值是否相等，默认会调用对象的**eq()**方法

25. 求出列表所有奇数并构造新列表

```
a = [1,2,3,4,5,6,7,8,9,10]
res = [ i for i in a if i%2==1]
print(res)
```

26. 用一行python代码写出1+2+3+10248

```
from functools import reduce
#1.使用sum内置求和函数
num = sum([1,2,3,10248])
print(num)
#2.reduce 函数
num1 = reduce(lambda x,y :x+y,[1,2,3,10248])
print(num1)
```

27. Python中变量的作用域？ （变量查找顺序）

函数作用域的**LEGB**顺序

1.什么是**LEGB?**

L： **local** 函数内部作用域

E: enclosing 函数内部与内嵌函数之间

G: global 全局作用域

B： **build-in** 内置作用

python在函数里面的查找分为**4**种，称之为**LEGB**，也正是按照这是顺序来查找的

28. 字符串 `"123"` 转换成 `123`，不使用内置api，例如`int()`

方法一： 利用 `str` 函数

```
def atoi(s):
    num = 0
    for v in s:
        for j in range(10):
            if v == str(j):
                num = num * 10 + j
    return num
```

方法二： 利用 `ord` 函数

```
def atoi(s):
    num = 0
    for v in s:
        num = num * 10 + ord(v) -ord('0')
    return num
```

方法三：利用 `eval` 函数

```
def atoi(s):
    num = 0
    for v in s:
        t = "%s * 1" % v
        n = eval(t)
        num = num * 10 + n
    return num
```

方法四：结合方法二，使用 `reduce`，一行解决

```
from functools import reduce
def atoi(s):
    return reduce(lambda num, v: num * 10 + ord(v) - ord('0'), s, 0)
```

29. Given an array of integers

给定一个整数数组和一个目标值，找出数组中和为目标值的两个数。你可以假设每个输入只对应一种答案，且同样的元素不能被重复利用。示例:给定**nums = [2,7,11,15],target=9** 因为 **nums[0]+nums[1] = 2+7 =9,**所以返回**[0,1]**

```
class Solution:
    def twoSum(self,nums,target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        d = {}
        size = 0
        while size < len(nums):
            if target-nums[size] in d:
                if d[target-nums[size]] <size:
                    return [d[target-nums[size]],size]
                else:
                    d[nums[size]] = size
                size = size +1
solution = Solution()
list = [2,7,11,15]
target = 9
nums =solution.twoSum(list,target)
print(nums)
```

```
class Solution(object):
    def twoSum(self, nums, target):
        for i in range(len(nums)):
            num = target - nums[i]
            if num in nums[i+1:]:
                return [i, nums.index(num,i+1)]
```

给列表中的字典排序：假设有如下**list**对象，**alist=[{"name":"a","age":20},{"name":"b","age":30},{"name":"c","age":25}],**将**alist**中的元素按照**age**从大到小排序 **alist=[{"name":"a","age":20},{"name":"b","age":30},{"name":"c","age":25}]**

```
alist_sort = sorted(alist,key=lambda e: e._getitem_('age'),reverse=True)
```

30. python代码实现删除一个list里面的重复元素

```
def distFunc1(a):
    """使用集合去重"""
    a = list(set(a))
    print(a)

def distFunc2(a):
    """将一个列表的数据取出放到另一个列表中，中间作判断"""
    list = []
    for i in a:
        if i not in list:
            list.append(i)
    #如果需要排序的话用sort
    list.sort()
    print(list)

def distFunc3(a):
    """使用字典"""
    b = {}
    b = b.fromkeys(a)
    c = list(b.keys())
    print(c)

if__name__== "_main__":
    a = [1,2,4,2,4,5,7,10,5,5,7,8,9,0,3]
    distFunc1(a)
    distFunc2(a)
    distFunc3(a)
```

31. 统计一个文本中单词频次最高的10个单词?

```
import re

# 方法一
def test(filepath):

    distone = {}
```

```
    with open(filepath) as f:
        for line in f:
            line = re.sub("\W+", " ", line)
            lineone = line.split()
            for keyone in lineone:
                if not distone.get(keyone):
                    distone[keyone] = 1
                else:
                    distone[keyone] += 1
    num_ten = sorted(distone.items(), key=lambda x:x[1], reverse=True)[:10]
    num_ten =[x[0] for x in num_ten]
    return num_ten


# 方法二
# 使用 built-in 的 Counter 里面的 most_common
import re
from collections import Counter


def test2(filepath):
    with open(filepath) as f:
        return list(map(lambda c: c[0], Counter(re.sub("\W+", " ",
f.read()).split()).most_common(10)))
```

32.请写出一个函数满足以下条件

该函数的输入是一个仅包含数字的**list,**输出一个新的**list**，其中每一个元素要满足以下条件：

1、该元素是偶数

2、该元素在原**list**中是在偶数的位置**(index**是偶数**)**

```
def num_list(num):
    return [i for i in num if i %2 ==0 and num.index(i)%2==0]

num = [0,1,2,3,4,5,6,7,8,9,10]
result = num_list(num)
print(result)
```

33.使用单一的列表生成式来产生一个新的列表

该列表只包含满足以下条件的值，元素为原始列表中偶数切片

```
list_data = [1,2,5,8,10,3,18,6,20]
res = [x for x in list_data[::2] if x %2 ==0]
print(res)
```

34.用一行代码生成[1,4,9,16,25,36,49,64,81,100]

```
[x * x for x in range(1,11)]
```

35.输入某年某月某日，判断这一天是这一年的第几天?

```
import datetime

y = int(input("请输入4位数字的年份:"))
m = int(input("请输入月份:"))
d = int(input("请输入是哪一天"))

targetDay = datetime.date(y,m,d)
dayCount = targetDay - datetime.date(targetDay.year -1,12,31)
print("%s是 %s年的第%s天。"%(targetDay,y,dayCount.days))
```

36.两个有序列表，l1,l2，对这两个列表进行合并不可使用extend

```
def loop_merge_sort(l1,l2):
    tmp = []
    while len(l1)>0 and len(l2)>0:
        if l1[0] <l2[0]:
            tmp.append(l1[0])
            del l1[0]
        else:
            tmp.append(l2[0])
            del l2[0]
    while len(l1)>0:
        tmp.append(l1[0])
        del l1[0]
    while len(l2)>0:
        tmp.append(l2[0])
        del l2[0]
    return tmp
```

37.给定一个任意长度数组，实现一个函数

让所有奇数都在偶数前面，而且奇数升序排列，偶数降序排序，如字符串**'1982376455',**变成**'1355798642'**

```
# 方法一
def func1(l):
    if isinstance(l, str):
        l = [int(i) for i in l]
    l.sort(reverse=True)
    for i in range(len(l)):
        if l[i] % 2 > 0:
            l.insert(0, l.pop(i))
    print(''.join(str(e) for e in l))

# 方法二
def func2(l):
    print("".join(sorted(l, key=lambda x: int(x) % 2 == 0 and 20 - int(x) or
int(x))))
```

38.写一个函数找出一个整数数组中，第二大的数

```
def find_second_large_num(num_list):
    """
    找出数组第2大的数字
```

```
    """
    # 方法一
    # 直接排序，输出倒数第二个数即可
    tmp_list = sorted(num_list)
    print("方法一\nSecond_large_num is :", tmp_list[-2])

    # 方法二
    # 设置两个标志位一个存储最大数一个存储次大数
    # two 存储次大值，one 存储最大值，遍历一次数组即可，先判断是否大于 one，若大于将 one 的值给 two，将 num_list[i] 的值给 one，否则比较是否大于two，若大于直接将 num_list[i] 的值给two，否则pass
    one = num_list[0]
    two = num_list[0]
    for i in range(1,len(num_list)):
        if num_list[i] > one:
            two = one
            one = num_list[i]
        elif num_list[i] > two:
            two = num_list[i]
    print("方法二\nSecond_large_num is :", two)

    # 方法三
    # 用 reduce 与逻辑符号 (and, or)
    # 基本思路与方法二一样，但是不需要用 if 进行判断。
    from functools import reduce
    num = reduce(lambda ot, x: ot[1] < x and (ot[1], x) or ot[0] < x and (x, ot[1]) or ot, num_list, (0, 0))[0]
    print("方法三\nSecond_large_num is :", num)


if__name__== '__main___':
    num_list = [34, 11, 23, 56, 78, 0, 9, 12, 3, 7, 5]
    find_second_large_num(num_list)
```

39.阅读一下代码他们的输出结果是什么?

```
def multi():
    return [lambda x : i*x for i in range(4)]
print([m(3) for m in multi()])
```

正确答案是**[9,9,9,9]**，而不是**[0,3,6,9]**产生的原因是**Python**的闭包的后期绑定导致的，这意味着在闭包中的变量是在内部函数被调用的时候被查找的，因为，最后函数被调用的时候，**for**循环已经完成，**i** 的值最后是**3,**因此每一个返回值的**i**都是**3,**所以最后的结果是**[9,9,9,9]**

40.统计一段字符串中字符出现的次数

```
# 方法一
def count_str(str_data):
    """定义一个字符出现次数的函数"""
    dict_str = {}
    for i in str_data:
        dict_str[i] = dict_str.get(i, 0) +1
    return dict_str
dict_str =count_str("AAABBCCAC")
str_count_data = ""
```

```
for k, v in dict_str.items():
    str_count_data += k + str(v)
print(str_count_data)

# 方法二
from collections import Counter

print("".join(map(lambda x: x[0] + str(x[1]),
Counter("AAABBCCAC").most_common())))
```

41. super函数的具体用法和场景

https://python3-cookbook.readthedocs.io/zh_CN/latest/c08/p07_calling_method_on_parent_class.html

Python高级

元类

42. Python中类方法、类实例方法、静态方法有何区别?

类方法: 是类对象的方法，在定义时需要在上方使用 **@classmethod** 进行装饰,形参为**cls**，表示类对象，类对象和实例对象都可调用

类实例方法: 是类实例化对象的方法,只有实例对象可以调用，形参为**self**,指代对象本身;

静态方法: 是一个任意函数，在其上方使用 **@staticmethod** 进行装饰，可以用对象直接调用，静态方法实际上跟该类没有太大关系

43. 遍历一个object的所有属性，并print每一个属性名?

```
class Car:
    def __init__(self,name,loss): # loss [价格，油耗，公里数]
        self.name = name
        self.loss = loss

    def getName(self):
```

```
        return self.name

    def getPrice(self):
        # 获取汽车价格
        return self.loss[0]

    def getLoss(self):
        # 获取汽车损耗值
        return self.loss[1] * self.loss[2]

Bmw = Car("宝马",[60,9,500]) # 实例化一个宝马车对象
print(getattr(Bmw,"name")) #使用getattr()传入对象名字,属性值。
print(dir(Bmw)) # 获Bmw所有的属性和方法
```

44.写一个类，并让它尽可能多的支持操作符?

```
class Array:
    __list = []

    def__init__(self):
        print "constructor"

    def__del__(self):
        print "destruct"

    def__str__(self):
        return "this self-defined array class"

    def__getitem__(self,key):
        return self.__list[key]

    def__len__(self):
        return len(self.__list)

    def Add(self,value):
        self.__list.append(value)

    def Remove(self,index):
        del self.__list[index]

    def DisplayItems(self):
        print "show all items---"
        for item in self.__list:
            print item
```

45.介绍Cython，Pypy Cpython Numba各有什么缺点

Cython

46.请描述抽象类和接口类的区别和联系

1.抽象类：　规定了一系列的方法，并规定了必须由继承类实现的方法。由于有抽象方法的存在，所以抽象类不能实例化。可以将抽象类理解为毛坯房，门窗，墙面的样式由你自己来定，所以抽象类与作为基　类的普通类的区别在于约束性更强

2.接口类：与抽象类很相似，表现在接口中定义的方法，必须由引用类实现，但他与抽象类的根本区别在于用途：与不同个体间沟通的规则，你要进宿舍需要有钥匙，这个钥匙就是你与宿舍的接口，你的舍友也有这个接口，所以他也能进入宿舍，你用手机通话，那么手机就是你与他人交流的接口

3.区别和关联：

1.接口是抽象类的变体，接口中所有的方法都是抽象的，而抽象类中可以有非抽象方法，抽象类是声明方法的存在而不去实现它的类

2.接口可以继承，抽象类不行

3.接口定义方法，没有实现的代码，而抽象类可以实现部分方法

4.接口中基本数据类型为**static**而抽象类不是

47.Python中如何动态获取和设置对象的属性?

```
if hasattr(Parent, 'x'):
    print(getattr(Parent, 'x'))
    setattr(Parent, 'x',3)
print(getattr(Parent,'x'))
```

内存管理与垃圾回收机制

48.哪些操作会导致Python内存溢出，怎么处理?

49.关于Python内存管理,下列说法错误的是 B

A,变量不必事先声明　　**B,**变量无须先创建和赋值而直接使用

C,变量无须指定类型　　**D,**可以使用**del**释放资源

50.Python的内存管理机制及调优手段?

内存管理机制**:** 引用计数、垃圾回收、内存池

引用计数：引用计数是一种非常高效的内存管理手段，当一个**Python**对象被引用时其引用计数增加**1,**

当其不再被一个变量引用时则计数减**1,**当引用计数等于**0**时对象被删除。弱引用不会增加引用计数

垃圾回收：

1.引用计数

引用计数也是一种垃圾收集机制，而且也是一种最直观、最简单的垃圾收集技术。当**Python**的某个对象的引用计数降为**0**时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。比如某个新建对象，它被分配给某个引用，对象的引用计数变为**1**，如果引用被删除，对象的引用计数为**0,**那么该对象就可以被垃圾回收。不过如果出现循环引用的话，引用计数机制就不再起有效的作用了。

2.标记清除

https://foofish.net/python-gc.html

调优手段

1.手动垃圾回收

2.调高垃圾回收阈值

3.避免循环引用

51.内存泄露是什么？如何避免？

内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

有`__del__()`函数的对象间的循环引用是导致内存泄露的主凶。不使用一个对象时使用**: del object** 来删除一个对象的引用计数就可以有效防止内存泄露问题。

通过**Python**扩展模块**gc** 来查看不能回收的对象的详细信息。

可以通过 **sys.getrefcount(obj)** 来获取对象的引用计数，并根据返回值是否为**0**来判断是否内存泄露

函数

52.python常见的列表推导式？

[表达式 **for** 变量 **in** 列表**]** 或者 **[**表达式 **for** 变量 **in** 列表 **if** 条件**]**

53.简述read、readline、readlines的区别？

read	读取整个文件
readline	读取下一行
readlines	读取整个文件到一个迭代器以供我们遍历

54.什么是Hash（散列函数）？

散列函数（英语：**Hash function**）又称**散列算法**、**哈希函数**，是一种从任何一种数据中创建小的数字**“**指纹**”**的方法。散列函数把消息或数据压缩成摘要，使得数据量变小，将数据的格式固定下来。该函数将数据打乱混合，重新创建一个叫做**散列值**（**hash values**，**hash codes**，**hash sums**，或**hashes**）的指纹。散列值通常用一个短的随机字母和数字组成的字符串来代表

55.python函数重载机制？

函数重载主要是为了解决两个问题。

1。可变参数类型。

2。可变参数个数。

另外，一个基本的设计原则是，仅仅当两个函数除了参数类型和参数个数不同以外，其功能是完全相同的，此时才使用函数重载，如果两个函数的功能其实不同，那么不应当使用重载，而应当使用一个名字不同的函数。

好吧，那么对于情况 **1** ，函数功能相同，但是参数类型不同，**python** 如何处理？答案是根本不需要处理，因为 **python** 可以接受任何类型的参数，如果函数的功能相同，那么不同的参数类型在 **python** 中很可能是相同的代码，没有必要做成两个不同函数。

那么对于情况 **2** ，函数功能相同，但参数个数不同，**python** 如何处理？大家知道，答案就是缺省参数。对那些缺少的参数设定为缺省参数即可解决问题。因为你假设函数功能相同，那么那些缺少的参数终归是需要用的。

好了，鉴于情况 **1** 跟 情况 **2** 都有了解决方案，**python** 自然就不需要函数重载了。

56.写一个函数找出一个整数数组中，第二大的数

设置两个标志位一个存储最大数，一个存储次大数 **two** 存储次大值，**one** 存储最大值，遍历一次数组即可，先判断是否大于 **one**，若大于将**one** 的值给 **two**，将 **num_list[i]** 的值给 **one**，否则比较是否大于 **two**，若大于直接将 **num_list[i]**
的值给**two**，否则**pass**。

```
def find_second_large_num(num_list):
    one = num_list[0]
    two = num_list[0]
    for i in range(1, len(num_list)):
        if num_list[i] > one:
            two = one
            one = num_list[i]
        elif num_list[i] > two:
            two = num_list[i]
    print("第二个大的数是 :", two)

num_list = [34, 11, 23, 56, 78, 0, 9, 12, 3, 7, 5]
find_second_large_num(num_list)
```

57.手写一个判断时间的装饰器

```
import datetime


class TimeException(Exception):
    def_init_(self, exception_info):
        super()._init__()
        self.info = exception_info

    def_str_(self):
        return self.info


def timecheck(func):
    def wrapper(*args, **kwargs):
        if datetime.datetime.now().year ==2019:
            func(*args, **kwargs)
        else:
            raise TimeException("函数已过时")

    return wrapper


@timecheck
def test(name):
    print("Hello {}, 2019 Happy".format(name))


if___name__== "_main_":
    test("backbp")
```

58.使用Python内置的filter()方法来过滤?

```
list(filter(lambda x: x % 2 == 0, range(10)))
```

59.编写函数的4个原则

1.函数设计要尽量短小

2.函数声明要做到合理、简单、易于使用

3.函数参数设计应该考虑向下兼容

4.一个函数只做一件事情，尽量保证函数语句粒度的一致性

60.函数调用参数的传递方式是值传递还是引用传递?

Python的参数传递有：位置参数、默认参数、可变参数、关键字参数。

函数的传值到底是值传递还是引用传递、要分情况：

不可变参数用值传递：像整数和字符串这样的不可变对象，是通过拷贝进行传递的，因为你无论如何都不可能在原处改变不可变对象。

可变参数是引用传递：比如像列表，字典这样的对象是通过引用传递、和**C**语言里面的用指针传递数组很相似，可变对象能在函数内部改变。

61.如何在function里面设置一个全局变量

```
globals() # 返回包含当前作用余全局变量的字典。
global 变量 设置使用全局变量
```

62.对缺省参数的理解？

缺省参数指在调用函数的时候没有传入参数的情况下，调用默认的参数，在调用函数的同时赋值时，所传入的参数会替代默认参数。

***args**是不定长参数，它可以表示输入参数是不确定的，可以是任意多个。

****kwargs**是关键字参数，赋值的时候是以键值对的方式，参数可以是任意多对在定义函数的时候

不确定会有多少参数会传入时，就可以使用两个参数

64.带参数的装饰器?

带定长参数的装饰器

```
def new_func(func):
    def wrappedfun(username, passwd):
        if username == 'root' and passwd == '123456789':
            print('通过认证')
            print('开始执行附加功能')
            return func()
        else:
            print('用户名或密码错误')
            return
    return wrappedfun

@new_func
def origin():
    print('开始执行函数')
origin('root','123456789')
```

带不定长参数的装饰器

```
def new_func(func):
    def wrappedfun(*parts):
        if parts:
            counts = len(parts)
            print('本系统包含 ', end='')
            for part in parts:
                print(part, ' ',end='')
            print('等', counts, '部分')
            return func()
        else:
            print('用户名或密码错误')
            return func()
   return wrappedfun
```

65.为什么函数名字可以当做参数用?

Python中一切皆对象，函数名是函数在内存中的空间，也是一个对象

66.Python中pass语句的作用是什么?

在编写代码时只写框架思路，具体实现还未编写就可以用**pass**进行占位，是程序不报错，不会进行任何操作。

67.有这样一段代码，print c会输出什么，为什么?

```
a = 10
b = 20
c = [a]
a = 15
```

答：**10**对于字符串，数字，传递是相应的值

68.交换两个变量的值?

```
a, b = b, a
```

69.map函数和reduce函数?

```
map(lambda x: x * x, [1, 2, 3, 4])   # 使用 lambda
# [1, 4, 9, 16]
reduce(lambda x, y: x * y, [1, 2, 3, 4])  # 相当于 ((1 * 2) * 3) * 4
# 24
```

70.回调函数，如何通信的?

回调函数是把函数的指针(地址)作为参数传递给另一个函数，将整个函数当作一个对象，赋值给调用的函数。

71.Python主要的内置数据类型都有哪些？ print dir(‘a ’) 的输出?

内建类型：布尔类型，数字，字符串，列表，元组，字典，集合

输出字符串**'a'**的内建方法

72. map(lambda x:xx，[y for y in range(3)])的输出?

```
[0, 1, 4]
```

73. hasattr() getattr() setattr() 函数使用详解?

hasattr(object,name)函数**:**

判断一个对象里面是否有**name**属性或者**name**方法，返回**bool**值，有**name**属性（方法）返回**True**，否则返回**False**。

```
class function_demo(object):
    name = 'demo'
    def run(self):
        return "hello function"
functiondemo = function_demo()
res = hasattr(functiondemo, "name") # 判断对象是否有name属性，True
res = hasattr(functiondemo, "run") # 判断对象是否有run方法，True
res = hasattr(functiondemo, "age") # 判断对象是否有age属性，False
print(res)
```

getattr(object, name[,default])函数:

获取对象**object**的属性或者方法，如果存在则打印出来，如果不存在，打印默认值，默认值可选。注意：如果返回的是对象的方法，则打印结果是：方法的内存地址，如果需要运行这个方法，可以在后面添加括号**()**.

```
functiondemo = function_demo()
getattr(functiondemo, "name")# 获取name属性，存在就打印出来 --- demo
getattr(functiondemo, "run") # 获取run 方法，存在打印出方法的内存地址
getattr(functiondemo, "age") # 获取不存在的属性，报错
getattr(functiondemo, "age", 18)# 获取不存在的属性，返回一个默认值
```

setattr(object, name, values)函数:

给对象的属性赋值，若属性不存在，先创建再赋值

```
class function_demo(object):
    name = "demo"
    def run(self):
        return "hello function"
functiondemo = function_demo()
res = hasattr(functiondemo, "age") # 判断age属性是否存在，False
print(res)
setattr(functiondemo, "age", 18) # 对age属性进行赋值，无返回值
res1 = hasattr(functiondemo, "age") # 再次判断属性是否存在，True
```

综合使用

```
class function_demo(object):
    name = "demo"
    def run(self):
        return "hello function"
functiondemo = function_demo()
res = hasattr(functiondemo, "addr") # 先判断是否存在
if res:
    addr = getattr(functiondemo, "addr")
    print(addr)
else:
    addr = getattr(functiondemo, "addr", setattr(functiondemo, "addr", "北京首都"))
    print(addr)
```

74.一句话解决阶乘函数?

```
reduce(lambda x,y : x*y,range(1,n+1))
```

75.什么是lambda函数? 有什么好处?

lambda 函数是一个可以接收任意多个参数**(**包括可选参数**)**并且返回单个表达式值的函数

1.lambda函数比较轻便，即用即仍，很适合需要完成一项功能，但是此功能只在此一处使用，连名字都很随意的情况下

2.匿名函数，一般用来给**filter**，**map**这样的函数式编程服务

3.作为回调函数，传递给某些应用，比如消息处理

76.递归函数停止的条件?

递归的终止条件一般定义在递归函数内部，在递归调用前要做一个条件判断，根据判断的结果选择是继续调用自身，还是**return**，，返回终止递归。

终止的条件：判断递归的次数是否达到某一限定值

2.判断运算的结果是否达到某个范围等，根据设计的目的来选择

77.下面这段代码的输出结果将是什么？请解释。

```
def multipliers():
    return [lambda x: i *x for i in range(4)]
    print([m(2) for m in multipliers()])
```

上面代码的输出结果是**[6,6,6,6]**，不是我们想的**[0,2,4,6]**

你如何修改上面的**multipliers**的定义产生想要的结果?

上述问题产生的原因是**python**闭包的延迟绑定。这意味着内部函数被调用时，参数的值在闭包内进行查找。因此，当任何由**multipliers()**返回的函数被调用时**,i**的值将在附近的范围进行查找。那时，不管返回的函数是否被调用，**for**循环已经完成，**i**被赋予了最终的值**3.**

```
def multipliers():
    for i in range(4):
        yield lambda x: i *x
```

```
def multipliers():
    return [lambda x,i = i: i*x for i in range(4)]
```

78.什么是lambda函数？它有什么好处？写一个匿名函数求两个数的和

lambda函数是匿名函数，使用**lambda**函数能创建小型匿名函数，这种函数得名于省略了用**def**声明函数的标准步骤

设计模式

79.对设计模式的理解，简述你了解的设计模式？

设计模式是经过总结，优化的，对我们经常会碰到的一些编程问题的可重用解决方案。一个设计模式并不像一个类或一个库那样能够直接作用于我们的代码，反之，设计模式更为高级，它是一种必须在特定情形下实现的一种方法模板。

常见的是工厂模式和单例模式

80.请手写一个单例

```
#python2
class A(object):
    __instance = None
    def_new_(cls,*args,**kwargs):
        if cls._instance is None:
            cls._instance = objecet._new_(cls)
            return cls.__instance
        else:
            return cls.__instance
```

81.单例模式的应用场景有那些？

单例模式应用的场景一般发现在以下条件下：
资源共享的情况下，避免由于资源操作时导致的性能或损耗等，如日志文件，应用配置。
控制资源的情况下，方便资源之间的互相通信。如线程池等，**1,**网站的计数器 **2,**应用配置 **3.**多线程池 **4** 数据库配置 数据库连接池 **5.**应用程序的日志应用**...**

82.用一行代码生成[1,4,9,16,25,36,49,64,81,100]

```
print([x*x for x in range(1, 11)])
```

83.对装饰器的理解，并写出一个计时器记录方法执行性能的装饰器?

装饰器本质上是一个**callable object**，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。

```
import time
from functools import wraps

def timeit(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.clock()
        ret = func(*args, **kwargs)
        end = time.clock()
        print('used:',end-start)
        return ret

    return wrapper
@timeit
def foo():
    print('in foo()'foo())
```

84.解释以下什么是闭包?

在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将这个函数以及用到的一些变量称之为闭包。

85.函数装饰器有什么作用?

装饰器本质上是一个**callable object**，它可以在让其他函数在不需要做任何代码的变动的前提下增加额外的功能。装饰器的返回值也是一个函数的对象，它经常用于有切面需求的场景。比如：插入日志，性能测试，事务处理，缓存。权限的校验等场景，有了装饰器就可以抽离出大量的与函数功能本身无关的雷同代码并发并继续使用。
详细参考：**https://manjusaka.itscoder.com/2018/02/23/something-about-decorator/**

86.生成器，迭代器的区别?

迭代器是遵循迭代协议的对象。用户可以使用 **iter()** 以从任何序列得到迭代器（如 **list, tuple, dictionary, set**等）。另一个方法则是创建一个另一种形式的迭代器——**generator**。要获取下一个元素，则使用成员函数 **next()**（**Python 2**）或函数 **next() function** （**Python 3**） 。当没有元素时，则引发 **StopIteration** 此例外。若要实现自己的迭代器，则只要实现 **next()**（**Python 2**）或 `__next__()`（**Python 3**）

生成器（**Generator**），只是在需要返回数据的时候使用**yield**语句。每次**next()**被调用时，生成器会返回它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）

区别： 生成器能做到迭代器能做的所有事，而且因为自动创建**iter()**和**next()**方法，生成器显得特别简洁，而且生成器也是高效的，使用生成器表达式取代列表解析可以同时节省内存。除了创建和保存程序 状态的自动方法，当发生器终结时，还会自动抛出**StopIteration**异常。

官方介绍：**https://docs.python.org/3/tutorial/classes.html#iterators**

87. X是什么类型?

```
X= (i for i in range(10))
```

X是 **generator**类型

88. 请用一行代码 实现将1-N 的整数列表以3为单位分组

```
N =100
print ([[x for x in range(1,100)] [i:i+3] for i in range(0,100,3)])
```

89. Python中yield的用法?

yield就是保存当前程序执行状态。你用**for**循环的时候，每次取一个元素的时候就会计算一次。用**yield**的函数叫**generator,**和**iterator**一样，它的好处是不用一次计算所有元素，而是用一次算一次，可以节省很多空间，**generator**每次计算需要上一次计算结果，所以用**yield,**否则一**return**，上次计算结果就没了

面向对象

90. Python中的可变对象和不可变对象?

不可变对象，该对象所指向的内存中的值不能被改变。当改变某个变量时候，由于其所指的值不能被改变，相当于把原来的值复制一份后再改变，这会开辟一个新的地址，变量再指向这个新的地址。

可变对象，该对象所指向的内存中的值可以被改变。变量（准确的说是引用）改变后，实际上其所指的值直接发生改变，并没有发生复制行为，也没有开辟出新的地址，通俗点说就是原地改变。

Pyhton中，数值类型**(int** 和**float)**，字符串**str**、元祖**tuple**都是不可变类型。而列表**list**、字典**dict**、集合**set**是可变类型

91. Python的魔法方法

魔法方法就是可以给你的类增加魔力的特殊方法，如果你的对象实现（重载）了这些方法中的某一个，那么这个方法就会在特殊的情况下被**Python**所调用，你可以定义自己想要的行为，而这一切都是自动发生的，它们经常是两个下划线包围来命名的（比如 `__init__`,`__len__`),**Python**的魔法方法是非常强大的所以了解其使用方法也变得尤为重要**!**

`__init__`构造器，当一个实例被创建的时候初始化的方法，但是它并不是实例化调用的第一个方法。

`__new__`才是实例化对象调用的第一个方法，它只取下**cls**参数，并把其他参数传给 `__init__`.

`__new__`很少使用，但是也有它适合的场景，尤其是当类继承自一个像元祖或者字符串这样不经常改变的类型的时候。

`__call__`让一个类的实例像函数一样被调用

`__getitem__`定义获取容器中指定元素的行为，相当于**self[key]**

`__getattr__`定义当用户试图访问一个不存在属性的时候的行为。

`__setattr__`定义当一个属性被设置的时候的行为

`__getattribute__`定义当一个属性被访问的时候的行为

92. 面向对象中怎么实现只读属性?

将对象私有化，通过共有方法提供一个读取数据的接口

```
class person:
    def_init_(self, x):
        self._age = 10
    def age(self):
        return self._age
t = person(22)
# t._age =100
print(t.age())
```

最好的方法

```
class MyCls(object):
    __weight = 50

    @property
    def weight(self):
        return self.__weight
```

93. 谈谈你对面向对象的理解?

面向对象是相当于面向过程而言的，面向过程语言是一种基于功能分析的，以算法为中心的程序设计方法，而面向对象是一种基于结构分析的，以数据为中心的程序设计思想。在面向对象语言中有一个很重要的东西，叫做类。面向对象有三大特性：封装、继承、多态。

正则表达式

94.a = “abbbccc”，用正则匹配为abccc,不管有多少b，就出现一次?

思路：不管有多少个b替换成一个

```
re.sub(r'b+', 'b', a)
```

95. Python字符串查找和替换?

a、str.find()：正序字符串查找函数
函数原型：
str.find(substr [,pos_start [,pos_end]])
返回str中第一次出现的substr的第一个字母的标号，如果str中没有substr则返回-1，也就是说从左边算起的第一次出现的substr的首字母标号。

参数说明：
str： 代 表 原 字 符 串
substr：代表要查找的字符串
pos_start：代表查找的开始位置，默认是从下标0开始查找
pos_end：代表查找的结束位置

例子：
'aabbcc.find('bb')' # 2

b、str.index()：正序字符串查找函数
index()函数类似于find()函数，在Python中也是在字符串中查找子串第一次出现的位置，跟find()不同的是，未找到则抛出异常。

函数原型：
str.index(substr [, pos_start, [pos_end]])

参数说明：
str： 代 表 原 字 符 串
substr：代表要查找的字符串
pos_start：代表查找的开始位置，默认是从下标0开始查找
pos_end：代表查找的结束位置

例子：
'acdd l1 23'.index(' ') # 4

c、str.rfind()：倒序字符串查找函数

函数原型：
str.rfind(substr [, pos_start [,pos_ end]])
返回str中最后出现的substr的第一个字母的标号，如果str中没有substr则返回-1，也就是说从右边算起的第一次出现的substr的首字母标号。

参数说明：
str： 代 表 原 字 符 串
substr：代表要查找的字符串
pos_start：代表查找的开始位置，默认是从下标0开始查找

pos_end：代表查找的结束位置

例子：
'adsfddf'.rfind('d') # 5

d、str.rindex()：倒序字符串查找函数
rindex()函数类似于rfind()函数，在Python中也是在字符串中倒序查找子串最后一次出现的位置，跟rfind()不同的是，未找到则抛出异常。

函数原型：
str.rindex(substr [, pos_start, [pos_end]])

参数说明：
str：代表原字符串
substr：代表要查找的字符串
pos_start：代表查找的开始位置，默认是从下标0开始查找
pos_end：代表查找的结束位置

例子：
 'adsfddf'.rindex('d') # 5

e、使用re模块进行查找和替换：

函数	说明
re.match(pat, s)	只从字符串**s**的头开始匹配，比如**(‘123’, ‘12345’)**匹配上了，而**(‘123’,’01234’)**就是没有匹配上，没有匹配上返回**None**，匹配上返回**matchobject**
re.search(pat, s)	从字符串**s**的任意位置都进行匹配，比如**(‘123’,’01234’)**就是匹配上了，只要**s**只能存在符合**pat**的连续字符串就算匹配上了，没有匹配上返回**None**，匹配上返回**matchobject**
re.sub(pat,newpat,s)	**re.sub(pat,newpat,s)** 对字符串中**s**的包含的所有符合**pat**的连续字符串进行替换，如果**newpat**为**str,**那么就是替换为**newpat,**如果**newpat**是函数，那么就按照函数返回值替换。**sub**函数两个有默认值的参数分别是**count**表示最多只处理前几个匹配的字符串，默认为**0**表示全部处理；最后一个是**flags**，默认为**0**

f、使用replace()进行替换：
基本用法：对象.replace(rgExp,replaceText,max)

其中，rgExp和replaceText是必须要有的，max是可选的参数，可以不加。
rgExp是指正则表达式模式或可用标志的正则表达式对象，也可以是 String 对象或文字；
replaceText是一个String 对象或字符串文字；
max是一个数字。
对于一个对象，在对象的每个rgExp都替换成replaceText，从左到右最多max次。

s1='hello world'
s1.replace('world','liming')

96.用Python匹配HTML tag的时候，<.> 和 <.?> 有什么区别

第一个代表贪心匹配，第二个代表非贪心；
?在一般正则表达式里的语法是指的"零次或一次匹配左边的字符或表达式"相当于{0,1}
而当?后缀于*,+,?,{n},{n,},{n,m}之后，则代表非贪心匹配模式，也就是说，尽可能少的匹配左边的字符或表达式，这里是尽可能少的匹配.(任意字符)

所以：第一种写法是，尽可能多的匹配，就是匹配到的字符串尽量长，第二中写法是尽可能少的匹配，就是匹配到的字符串尽量短。
比如<tag>tag>tag>end，第一个会匹配<tag>tag>tag>,第二个会匹配<tag>。

97.正则表达式贪婪与非贪婪模式的区别?

贪婪模式：
定义：正则表达式去匹配时，会尽量多的匹配符合条件的内容标
识符：+，?，*，{n}，{n,}，{n,m}
匹配时，如果遇到上述标识符，代表是贪婪匹配，会尽可能多的去匹配内容

非贪婪模式：
定义：正则表达式去匹配时，会尽量少的匹配符合条件的内容　也就是说，一旦发现匹配符合要求，立马就匹配成功，而不会继续匹配下去(除非有g，开启下一组匹配)
标识符：+?，??，*?，{n}?，{n,}?，{n,m}?
可以看到，非贪婪模式的标识符很有规律，就是贪婪模式的标识符后面加上一个?

参考文章：https://dailc.github.io/2017/07/06/regularExpressionGreedyAndLazy.html

98.写出开头匹配字母和下划线，末尾是数字的正则表达式?

```
s1='_aai0efe00'
res=re.findall('^[a-zA-Z_]?[a-zA-Z0-9_]{1,}\d$',s1)
print(res)
```

99.怎么过滤评论中的表情?

思路：主要是匹配表情包的范围，将表情包的范围用空替换掉

```
import re
pattern = re.compile(u'[\uD800-\uDBFF][\uDC00-\uDFFF]')
pattern.sub('',text)
```

100. 简述Python里面search和match的区别

match()函数只检测字符串开头位置是否匹配，匹配成功才会返回结果，否则返回None；
search()函数会在整个字符串内查找模式匹配,只到找到第一个匹配然后返回一个包含匹配信息的对象,该对象可以通过调用group()方法得到匹配的字符串,如果字符串没有匹配，则返回None。

系统编程

101. 进程总结

进程：程序运行在操作系统上的一个实例，就称之为进程。进程需要相应的系统资源：内存、时间片、**pid**。
创建进程：
首先要导入**multiprocessing**中的**Process**：
创建一个**Process**对象**;**
创建**Process**对象时，可以传递参数**;**

```
p = Process(target=XXX,args=(tuple,),kwargs={key:value})
target = XXX 指定的任务函数，不用加(),
args=(tuple,)kwargs={key:value}给任务函数传递的参数
```

使用**start()**启动进程
结束进程
给子进程指定函数传递参数**Demo**

```
import os
from mulitprocessing import Process
import time

def pro_func(name,age,**kwargs):
    for i in range(5):
        print("子进程正在运行中，name=%s,age=%d,pid=%d"%(name,age,os.getpid()))
        print(kwargs)
        time.sleep(0.2)
if__name__=="_main_":
    #创建Process对象
    p = Process(target=pro_func,args=('小明',18),kwargs={'m':20})
    #启动进程
    p.start()
    time.sleep(1)
    #1秒钟之后，立刻结束子进程
    p.terminate()
    p.join()
```

注意：进程间不共享全局变量

进程之间的通信**-Queue**

在初始化**Queue()**对象时（例如**q=Queue(),**若在括号中没有指定最大可接受的消息数量，获数量为负值时，那么就代表可接受的消息数量没有上限一直到内存尽头）

Queue.qsize():返回当前队列包含的消息数量

Queue.empty():如果队列为空，返回**True**，反之**False**

Queue.full():如果队列满了，返回**True,**反之**False**

Queue.get([block[,timeout]]):获取队列中的一条消息，然后将其从队列中移除，

block默认值为**True**。

如果**block**使用默认值，且没有设置**timeout**（单位秒**),**消息队列如果为空，此时程序将被阻塞（停在读中状态），直到消息队列读到消息为止，如果设置了**timeout**，则会等待**timeout**秒，若还没读取到任何消息，则抛出**"Queue.Empty"**异常：

Queue.get_nowait()相当于**Queue.get(False)**

Queue.put(item,[block[,timeout]]):将**item**消息写入队列，**block**默认值为**True;**
如果**block**使用默认值，且没有设置**timeout**（单位秒），消息队列如果已经没有空间可写入，此时程序将被阻塞（停在写入状态），直到从消息队列腾出空间为止，如果设置了**timeout**，则会等待**timeout**秒，若还没空间，则抛出**"Queue.Full"**异常
如果**block**值为**False**，消息队列如果没有空间可写入，则会立刻抛出**"Queue.Full"**异常**;**
Queue.put_nowait(item):相当**Queue.put(item,False)**

进程间通信**Demo:**

```
from multiprocessing import Process.Queue
import os,time,random
#写数据进程执行的代码:
def write(q):
    for value in ['A','B','C']:
        print("Put %s to queue...",%value)
        q.put(value)
        time.sleep(random.random())
#读数据进程执行的代码
def read(q):
    while True:
        if not q.empty():
            value = q.get(True)
            print("Get %s from queue.",%value)
            time.sleep(random.random())
        else:
            break
if_name_=='_main__':
    #父进程创建Queue，并传给各个子进程
    q = Queue()
    pw =Process(target=write,args=(q,))
    pr = Process(target=read,args=(q,))
    #启动子进程pw ，写入:
    pw.start()
    #等待pw结束
    pw.join()
    #启动子进程pr，读取:
    pr.start()
    pr.join()
    #pr 进程里是死循环，无法等待其结束，只能强行终止:
    print('')
    print('所有数据都写入并且读完')
```

进程池Pool

```
#coding:utf-8
from multiprocessing import Pool
import os,time,random

def worker(msg):
    t_start = time.time()
    print("%s 开始执行，进程号为%d"%(msg,os.getpid()))
    # random.random()随机生成0-1之间的浮点数
    time.sleep(random.random()*2)
    t_stop = time.time()
    print(msg,"执行完毕，耗时%0.2f”%（t_stop-t_start))
```

```
po = Pool(3)#定义一个进程池，最大进程数3
for i in range(0,10):
    po.apply_async(worker,(i,))
print("---start--- ")
po.close()
po.join()
print("----end--- ")
```

进程池中使用**Queue**

如果要使用**Pool**创建进程，就需要使用**multiprocessing.Manager()**中的**Queue(),**而不是**multiprocessing.Queue(),**否则会得到如下的错误信息：

RuntimeError：**Queue objects should only be shared between processs through inheritance**

```
from multiprocessing import Manager,Pool
import os,time,random
def reader(q):
    print("reader 启动(%s),父进程为（%s)"%(os.getpid(),os.getpid()))
    for i in range(q.qsize()):
        print("reader 从Queue获取到消息:%s"%q.get(True))

def writer(q):
    print("writer 启动（%s),父进程为(%s)"%(os.getpid(),os.getpid()))
    for i ini "itcast":
        q.put(i)
if__name__== "_main_":
    print("(%s)start"%os.getpid())
    q = Manager().Queue()#使用Manager中的Queue
    po = Pool()
    po.apply_async(wrtier,(q,))
    time.sleep(1)
    po.apply_async(reader,(q,))
    po.close()
    po.join()
    print("(%s)End"%os.getpid())
```

102. 谈谈你对多进程，多线程，以及协程的理解，项目是否用?

这个问题被问的概念相当之大，
进程：一个运行的程序（代码）就是一个进程，没有运行的代码叫程序，进程是系统资源分配的最小单位，进程拥有自己独立的内存空间，所有进程间数据不共享，开销大。

线程**:** **cpu**调度执行的最小单位，也叫执行路径，不能独立存在，依赖进程存在，一个进程至少有一个线程，叫主线程，而多个线程共享内存（数据共享，共享全局变量**),**从而极大地提高了程序的运行效率。

协程**:** 是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和 栈，直接操中栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

103. Python异步使用场景有那些?

异步的使用场景**:**

1、 不涉及共享资源，获对共享资源只读，即非互斥操作

2、 没有时序上的严格关系

3、 不需要原子操作，或可以通过其他方式控制原子性

4、 常用于**IO**操作等耗时操作，因为比较影响客户体验和使用性能

5、 不影响主线程逻辑

104. 多线程共同操作同一个数据互斥锁同步?

```
import threading
import time
class MyThread(threading.Thread):
    def run(self):
        global num
        time.sleep(1)

        if mutex.acquire(1):
            num +=1
            msg = self.name + 'set num to ' +str(num)
            print msg
            mutex.release()
num = 0
mutex = threading.Lock()
def test():
    for i in range(5):
        t = MyThread()
        t.start()
if__name__=="__main__":
    test()
```

105. 什么是多线程竞争?

线程是非独立的，同一个进程里线程是数据共享的，当各个线程访问数据资源时会出现竞争状态即：数据几乎同步会被多个线程占用，造成数据混乱，即所谓的线程不安全

那么怎么解决多线程竞争问题？**---**锁

锁的好处： 确保了某段关键代码（共享数据资源）只能由一个线程从头到尾完整地执行能解决多线程资源竞争下的原子操作问题。

锁的坏处： 阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了

锁的致命问题**:** 死锁

106. 请介绍一下Python的线程同步?

一、 **setDaemon(False)**
当一个进程启动之后，会默认产生一个主线程，因为线程是程序执行的最小单位，当设置多线程时，主线程会创建多个子线程，在**Python**中，默认情况下就是**setDaemon(False),**主线程执行完自己的任务以后，就退出了，此时子线程会继续执行自己的任务，直到自己的任务结束。

例子

```
import threading
import time

def thread():
    time.sleep(2)
```

```
    print('---子线程结束---')

def main():
    t1 =threading.Thread(target=thread)
    t1.start()
    print('---主线程--结束')

if__name__=='_main_':
    main()
#执行结果
---主线程--结束
---子线程结束---
```

二、 **setDaemon（True)**

当我们使用**setDaemon(True)**时，这是子线程为守护线程，主线程一旦执行结束，则全部子线程被强制终止

例子

```
import threading
import time
def thread():
    time.sleep(2)
    print('---子线程结束---')
def main():
    t1 =threading.Thread(target=thread)
    t1.setDaemon(True)#设置子线程守护主线程
    t1.start()
    print('---主线程结束---')

if__name__=='_main_':
    main()
#执行结果
---主线程结束--- #只有主线程结束，子线程来不及执行就被强制结束
```

三、 **join**（线程同步)

join 所完成的工作就是线程同步，即主线程任务结束以后，进入堵塞状态，一直等待所有的子线程结束以后，主线程再终止。

当设置守护线程时，含义是主线程对于子线程等待**timeout**的时间将会杀死该子线程，最后退出程序，所以说，如果有**10**个子线程，全部的等待时间就是每个**timeout**的累加和，简单的来说，就是给每个子线程一个**timeou**的时间，让他去执行，时间一到，不管任务有没有完成，直接杀死。

没有设置守护线程时，主线程将会等待**timeout**的累加和这样的一段时间，时间一到，主线程结束，但是并没有杀死子线程，子线程依然可以继续执行，直到子线程全部结束，程序退出。

例子

```
import threading
import time

def thread():
    time.sleep(2)
    print('---子线程结束---')

def main():
    t1 = threading.Thread(target=thread)
```

```
    t1.setDaemon(True)
    t1.start()
    t1.join(timeout=1)#1 线程同步，主线程堵塞1s 然后主线程结束，子线程继续执行
                      #2 如果不设置timeout参数就等子线程结束主线程再结束
                      #3 如果设置了setDaemon=True和timeout=1主线程等待1s后会强制杀死
子线程，然后主线程结束
    print('---主线程结束---')

if__name__=='__main__':
    main()
```

107. 解释以下什么是锁，有哪几种锁?

锁(Lock)是python提供的对线程控制的对象。有互斥锁，可重入锁，死锁。

108. 什么是死锁?

若干子线程在系统资源竞争时，都在等待对方对某部分资源解除占用状态，结果是谁也不愿先解锁，互相干等着，程序无法执行下去，这就是死锁。

GIL锁 全局解释器锁

作用： 限制多线程同时执行，保证同一时间只有一个线程执行，所以cython里的多线程其实是伪多线程!

所以python里常常使用协程技术来代替多线程，协程是一种更轻量级的线程。

进程和线程的切换时由系统决定，而协程由我们程序员自己决定，而模块gevent下切换是遇到了耗时操作时才会切换

三者的关系：进程里有线程，线程里有协程。

109. 多线程交互访问数据，如果访问到了就不访问了?

怎么避免重读?

创建一个已访问数据列表，用于存储已经访问过的数据，并加上互斥锁，在多线程访问数据的时候先查看数据是否在已访问的列表中，若已存在就直接跳过。

110. 什么是线程安全，什么是互斥锁?

每个对象都对应于一个可称为'互斥锁'的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

同一进程中的多线程之间是共享系统资源的，多个线程同时对一个对象进行操作，一个线程操作尚未结束，另一线程已经对其进行操作，导致最终结果出现错误，此时需要对被操作对象添加互斥锁，保证每个线程对该对象的操作都得到正确的结果。

111. 说说下面几个概念：同步，异步，阻塞，非阻塞?

同步： 多个任务之间有先后顺序执行，一个执行完下个才能执行。

异步： 多个任务之间没有先后顺序，可以同时执行，有时候一个任务可能要在必要的时候获取另一个同时执行的任务的结果，这个就叫回调!

阻塞： 如果卡住了调用者，调用者不能继续往下执行，就是说调用者阻塞了。非

阻塞： 如果不会卡住，可以继续执行，就是说非阻塞的。

同步异步相对于多任务而言，阻塞非阻塞相对于代码执行而言。

112. 什么是僵尸进程和孤儿进程？怎么避免僵尸进程?

孤儿进程： 父进程退出，子进程还在运行的这些子进程都是孤儿进程，孤儿进程将被**init** 进程（进程号为**1**）所收养，并由**init** 进程对他们完成状态收集工作。

僵尸进程： 进程使用**fork** 创建子进程，如果子进程退出，而父进程并没有调用**wait** 获**waitpid** 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中的这些进程是僵尸进程。

避免僵尸进程的方法：

1.fork 两次用孙子进程去完成子进程的任务

2.用**wait()**函数使父进程阻塞

3.使用信号量，在**signal handler** 中调用**waitpid,**这样父进程不用阻塞

113. python中进程与线程的使用场景?

多进程适合在**CPU**密集操作（**cpu**操作指令比较多，如位多的的浮点运算）。

多线程适合在**IO**密性型操作（读写数据操作比多的的，比如爬虫）

114. 线程是并发还是并行，进程是并发还是并行?

线程是并发，进程是并行**;**

进程之间互相独立，是系统分配资源的最小单位，同一个线程中的所有线程共享资源。

115. 并行(parallel)和并发（concurrency)?

并行： 同一时刻多个任务同时在运行

不会在同一时刻同时运行，存在交替执行的情况。

实现并行的库有： **multiprocessing**

实现并发的库有**: threading**

程序需要执行较多的读写、请求和回复任务的需要大量的**IO**操作，**IO**密集型操作使用并发更好。

CPU运算量大的程序，使用并行会更好

116. IO密集型和CPU密集型区别?

IO密集型： 系统运行，大部分的状况是**CPU**在等 **I/O**（硬盘/内存）的读/写

CPU密集型： 大部分时间用来做计算，逻辑判断等**CPU**动作的程序称之**CPU**密集型。

117. python asyncio的原理?

asyncio这个库就是使用**python**的**yield**这个可以打断保存当前函数的上下文的机制， 封装好了**selector** 摆脱掉了复杂的回调关系

数据结构

118. 数组中出现次数超过一半的数字-Python版

思路：在遍历数组时保存两个值：一是数组中一个数字，一是次数。遍历下一个数字时，若它与之前保存的数字相同，则次数加**1**，否则次数减**1**；若次数为**0**，则保存下一个数字，并将次数置为**1**。遍历结束后，所保存的数字即为所求。然后再判断它是否符合条件即可。

时间复杂度:**O(N)**

```
def MoreThanHalfNum_Solution(numbers):
    len1 = len(numbers)
    if len1==0:
        return 0
    elif len1>=1:
        # 遍历每个元素，并记录次数；若与前一个元素相同，则次数加1，否则次数减1
        res = numbers[0] # 初始值
        count = 1 # 次数
        for i in range(1,len1):
            if count == 0:
                # 更新result的值为当前元素，并置次数为1
                res = numbers[i]
                count = 1
            elif numbers[i] == res:
                count += 1 # 相同则加1
            elif numbers[i] != res:
                count -= 1 # # 不同则加1
        # 判断res是否符合条件，即出现次数大于数组长度的一半
        counts = 0
        for j in range(len1):
            if numbers[j] == res:
                counts += 1
        if counts>len1//2:
            return res
        else:
            return 0
```

119. 求100以内的质数

```
#求100以内的全部素数
L=[]
for x in range(100):
    if x<2:
        continue
    for i in range(2,x):
        if x%i==0:
            break
    else:    #走到此处，x一定是素数
        L.append(x)
print("100以内的全部素数有：",L)
```

120. 无重复字符的最长子串-Python实现

可以利用滑动窗口、动态规划、双指针等方法解决该问题，详情见**LeetCode**第三题，这里给出双指针解题思路。

```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        res = 0
        mark = set()  # 用集合标明是否有出现重复字母
        r = 0  # 右指针
        for i in range(len(s)):
            if i != 0:
                mark.remove(s[i - 1])
            while r < len(s) and s[r] not in mark:  # 如果不满足条件说明r走到了s的尽头或r指向的元素
                mark.add(s[r])  # 将当前r指向的字母加入集合
                r += 1
            res = max(res, r - i)  # 在每一个位置更新最大值
        return res
```

121. 冒泡排序的思想?

冒泡排序是一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。

算法步骤：

- **1**、比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- **2**、对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
- **3**、针对所有的元素重复以上的步骤，除了最后一个。
- **4**、持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
def bubbleSort(arr):
    for i in range(1, len(arr)):
        for j in range(0, len(arr)-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

122. 快速排序的思想?

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 **n** 个项目要 **O(nlogn)** 次比较。在最坏状况下则需要 **O(n2)** 次比较，但这种状况并不常见。事实上，快速排序通常明显比其他 **O(nlogn)** 算法更快，因为它的内部循环（**inner loop**）可以在大部分的架构上很有效率地被实现出来。

快速排序使用分治法（**Divide and conquer**）策略来把一个串行（**list**）分为两个子串行（**sub-lists**）。快速排序又是一种分而治之思想在排序算法上的典型应用。本质上来看，快速排序应该算是在冒泡排序基础上的递归分治法。

算法步骤**:**

- **1**、从数列中挑出一个元素，称为 **“基准”（pivot）;**
- **2**、重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（**partition**）操作;
- **3**、递归地（**recursive**）把小于基准值元素的子数列和大于基准值元素的子数列排序;

```
def quickSort(arr, left=None, right=None):
    left = 0 if not isinstance(left,(int, float)) else left
    right = len(arr)-1 if not isinstance(right,(int, float)) else right
    if left < right:
        partitionIndex = partition(arr, left,right)
        quickSort(arr, left, partitionIndex-1)
        quickSort(arr, partitionIndex+1, right)
    return arr

def partition(arr, left, right):
    pivot = left
    index = pivot+1
    i = index
    while  i <= right:
        if arr[i] < arr[pivot]:
            swap(arr, i, index)
            index+=1
        i+=1
    swap(arr,pivot,index-1)
    return index-1

def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

123. 斐波那契数列

数列定义:

f 0 = f 1 = 1
f n = f (n-1) + f (n-2)

根据定义

速度很慢，另外**(**暴栈注意！⚠）`O(fibonacci n)`

```
def fibonacci(n):
    if n == 0 or n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

线性时间的

状态/循环

```
def fibonacci(n):
   a, b = 1, 1
   for _ in range(n):
       a, b = b, a + b
   return a
```

递归

```
def fibonacci(n):
    def fib(n_, s):
        if n_ == 0:
            return s[0]
        a, b = s
        return fib(n_ - 1, (b, a + b))
    return fib(n, (1, 1))
```

map(zipwith)

```
def fibs():
    yield 1
    fibs_ = fibs()
    yield next(fibs_)
    fibs__= fibs()
    for fib in map(lambad a, b: a + b, fibs_, fibs_):
        yield fib


def fibonacci(n):
    fibs_ = fibs()
    for _ in range(n):
        next(fibs_)
    return next(fibs)
```

做缓存

```
def cache(fn):
    cached = {}
    def wrapper(*args):
        if args not in cached:
            cached[args] = fn(*args)
        return cached[args]
    wrapper.__name__= fn.__name___
    return wrapper

@cache
def fib(n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```

利用 funtools.lru_cache 做缓存

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```

Logarithmic

矩阵

```
import numpy as np
def fibonacci(n):
    return (np.matrix([[0, 1], [1, 1]]) ** n)[1, 1]
```

不是矩阵

```
def fibonacci(n):
    def fib(n):
        if n == 0:
            return (1, 1)
        elif n == 1:
            return (1, 2)
        a, b = fib(n // 2 - 1)
        c = a + b
        if n % 2 == 0:
            return (a * a + b * b, c * c - a * a)
        return (c * c - a * a, b * b + c * c)
    return fib(n)[0]
```

124. 如何翻转一个单链表?

```
class Node:
    def_init_(self,data=None,next=None):
        self.data = data
        self.next = next

def rev(link):
    pre = link
    cur = link.next
    pre.next = None
    while cur:
        temp  = cur.next
        cur.next = pre
        pre = cur
        cur = tmp
    return pre

if__name__== '_main__':
    link=Node(1,Node(2,Node(3,Node(4,Node(5,Node(6,Node7,Node(8.Node(9))))))))
    root = rev(link)
    while root:
        print(roo.data)
        root = root.next
```

125. 青蛙跳台阶问题

一只青蛙要跳上**n**层高的台阶，一次能跳一级，也可以跳两级，请问这只青蛙有多少种跳上这个**n**层台阶的方法?

方法**1**：递归

设青蛙跳上**n**级台阶有**f(n)**种方法，把这**n**种方法分为两大类，第一种最后一次跳了一级台阶，这类共有**f(n-1)**种，第二种最后一次跳了两级台阶，这种方法共有**f(n-2)**种，则得出递推公式**f(n)=f(n-1) + f(n-2),**显然**f(1)=1,f(2)=2**，这种方法虽然代码简单，但效率低，会超出时间上限

```
class Solution:
    def climbStairs(self,n):
        if n ==1:
            return 1
        elif n==2:
            return 2
        else:
            return self.climbStairs(n-1) + self.climbStairs(n-2)
```

方法**2**：用循环来代替递归

```
class Solution:
    def climbStairs(self,n):
        if n==1 or n==2:
            return n
        a,b,c = 1,2,3
        for i in range(3,n+1):
            c = a+b
            a = b
            b = c
        return c
```

126. 写一个二分查找

过程：首先假设表中元素是按升序**(**降序**)**排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

```python
def binarySearch(alist, item):
    first = 0  #最小数下标
    last = len(alist) - 1 #最大数下标
    while first <= last:#只有当first小于High的时候证明中间有数
        mid = (first + last)//2 #中间数下标
        print(mid)
        if alist[mid] == item: #如果中间数下标等于item, 返回
            return mid
        elif alist[mid] >item: #如果item在中间数左边, 移动last下标
            last = mid - 1
        else:                   #如果val在中间数右边, 移动first下标
            first = mid+1
    return -1
```

127. Python实现一个Stack的数据结构

```python
"""栈  先进后出,后进先出"""""
class Stack(object):
    """创建一个新栈"""
    def_init__(self):
        """初始化栈"""
        self.data = []

    def push(self,item):
        """添加一个新的元素item到栈顶"""
        return self.data.append(item)
    def pop(self):
        """弹出栈顶元素"""
        return self.data.pop()
    def peek(self):
        """返回栈顶元素"""
        return self.data[-1]
        # return self.data[len(self.data)-1]
    def is_empty(self):
        """判断栈是否为空"""
        return self.data == []
    def size(self):
        """返回栈的元素个数"""
        return len(self.data)
```

128. Python实现一个Queue的数据结构

```python
"""队列  先进先出"""""
class Queue(object):
    def_init__(self):
        """创建一个空的队列"""
        self.data = []

    def enqueue(self,item):
        """往队列中添加一个item元素"""
        return self.data.insert(0,item)

    def dequeue(self):
        """从队列头部删除一个元素"""
        return self.data.pop()
```

```
    def is_empty(self):
        """判断一个队列是否为空"""
        return self.data == [ ]

    def size(self):
        """返回队列的大小"""
        return len(self.data)
```