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

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

63.Mysql怎么限制IP访问?

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.请写出一段代码用正则匹配出ip?

95.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)
```

96.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')
```

97.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>。

98.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

99.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)
```

100.正则表达式操作

101.请匹配出变量A 中的json字符串。

102.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)
```

103.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。

104.请写出匹配ip的Python正则表达式

105.Python里match与search的区别?

见103题

系统编程

106.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())
```

107.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。

108.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

109.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()
```

110.什么是多线程竞争?

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

111.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()
```

112.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。

113.什么是死锁?

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。

114.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。

115.什么是线程安全，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。

116.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。

117.什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

118.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）

119.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。

120.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

121.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。

122.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

网络编程

123.怎么实现强行关闭客户端和服务器之间的连接?

124.简述TCP和UDP的区别以及优缺点?

125.简述浏览器通过WSGI请求动态资源的过程?

浏览器发送的请求被Nginx监听到，Nginx根据请求的URL的PATH或者后缀把请求静态资源的分发到静态资源的目录，别的请求根据配置好的转发到相应端口。
实现了WSGI的程序会监听某个端口，监听到Nginx转发过来的请求接收后(一般用socket的recv来接收HTTP的报文)以后把请求的报文封装成`environ`的字典对象，然后再提供一个`start_response`的方法。把这两个对象当成参数传入某个方法比如`wsgi_app(environ, start_response)`或者实现了`__call__(self, environ, start_response)`方法的某个实例。这个实例再调用`start_response`返回给实现了WSGI的中间件，再由中间件返回给Nginx。

126.描述用浏览器访问www.baidu.com的过程

127.Post和Get请求的区别?

128.cookie 和session 的区别?

129.列出你知道的HTTP协议的状态码，说出表示什么意思?

130.请简单说一下三次握手和四次挥手?

131.说一下什么是tcp的2MSL?

132.为什么客户端在TIME-WAIT状态必须等待2MSL的时间?

133.说说HTTP和HTTPS区别?

134.谈一下HTTP协议以及协议头部中表示数据类型的字段?

135.HTTP请求方法都有什么?

136.使用Socket套接字需要传入哪些参数？

137.HTTP常见请求头?

138.七层模型?

139.url的形式?

Web

Flask

140.对Flask蓝图(Blueprint)的理解?

蓝图的定义

蓝图 /Blueprint 是Flask应用程序组件化的方法，可以在一个应用内或跨越多个项目共用蓝图。使用蓝图可以极大简化大型应用的开发难度，也为Flask扩展提供了一种在应用中注册服务的集中式机制。

蓝图的应用场景：

把一个应用分解为一个蓝图的集合。这对大型应用是理想的。一个项目可以实例化一个应用对象，初始化几个扩展，并注册一集合的蓝图。

以URL前缀和/或子域名，在应用上注册一个蓝图。URL前缀/子域名中的参数即成为这个蓝图下的所有视图函数的共同的视图参数（默认情况下）
在一个应用中用不同的URL规则多次注册一个蓝图。

通过蓝图提供模板过滤器、静态文件、模板和其他功能。一个蓝图不一定要实现应用或视图函数。

初始化一个Flask扩展时，在这些情况中注册一个蓝图。

蓝图的缺点：

不能在应用创建后撤销注册一个蓝图而不销毁整个应用对象。

使用蓝图的三个步骤

1.创建一个蓝图对象

```
blue = Blueprint("blue",__name__)
```

2.在这个蓝图对象上进行操作，例如注册路由、指定静态文件夹、注册模板过滤器...

```
@blue.route('/')
def blue_index():
    return "Welcome to my blueprint"
```

3.在应用对象上注册这个蓝图对象

```
app.register_blueprint(blue,url_prefix="/blue")
```

141.Flask 和 Django 路由映射的区别?

在django中，路由是浏览器访问服务器时，先访问的项目中的url，再由项目中的url找到应用中url，这些url是放在一个列表里，遵从从前往后匹配的规则。在flask中，路由是通过装饰器给每个视图函数提供的，而且根据请求方式的不同可以一个url用于不同的作用。

Django

142.什么是wsgi,uwsgi,uWSGI?

WSGI:

web服务器网关接口，是一套协议。用于接收用户请求并将请求进行初次封装，然后将请求交给web框架。

实现wsgi协议的模块：wsgiref,本质上就是编写一socket服务端，用于接收用户请求（django)

werkzeug,本质上就是编写一个socket服务端，用于接收用户请求(flask)

uwsgi:

与WSGI一样是一种通信协议，它是uWSGI服务器的独占协议，用于定义传输信息的类型。
uWSGI:

是一个web服务器，实现了WSGI的协议，uWSGI协议，http协议

143.Django、Flask、Tornado的对比?

1、 Django走的大而全的方向，开发效率高。它的MTV框架，自带的ORM,admin后台管理,自带的sqlite数据库和开发测试用的服务器，给开发者提高了超高的开发效率。
重量级web框架，功能齐全，提供一站式解决的思路，能让开发者不用在选择上花费大量时间。

自带ORM和模板引擎，支持jinja等非官方模板引擎。

自带ORM使Django和关系型数据库耦合度高，如果要使用非关系型数据库，需要使用第三方库

自带数据库管理app

成熟，稳定，开发效率高，相对于Flask，Django的整体封闭性比较好，适合做企业级网站的开发。
python web框架的先驱，第三方库丰富

2、 Flask 是轻量级的框架，自由，灵活，可扩展性强，核心基于Werkzeug WSGI工具 和jinja2 模板引擎

适用于做小网站以及web服务的API,开发大型网站无压力，但架构需要自己设计

与关系型数据库的结合不弱于Django，而与非关系型数据库的结合远远优于Django

3、 Tornado走的是少而精的方向，性能优越，它最出名的异步非阻塞的设计方式

Tornado的两大核心模块：

iostraem:对非阻塞的socket进行简单的封装

ioloop: 对I/O 多路复用的封装,它实现一个单例

144.CORS 和 CSRF的区别?

什么是CORS?

CORS是一个W3C标准,全称是"跨域资源共享"(Cross-origin resoure sharing).
它允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而客服了AJAX只能同源使用的限制。

什么是CSRF?

CSRF主流防御方式是在后端生成表单的时候生成一串随机token,内置到表单里成为一个字段，同时，将此串token置入session中。每次表单提交到后端时都会检查这两个值是否一致，以此来判断此次表单提交是否是可信的，提交过一次之后，如果这个页面没有生成CSRF token,那么token将会被清空,如果有新的需求，那么token会被更新。
攻击者可以伪造POST表单提交，但是他没有后端生成的内置于表单的token，session中没有token都无济于事。

145.Session,Cookie,JWT的理解

为什么要使用会话管理

众所周知，HTTP协议是一个无状态的协议，也就是说每个请求都是一个独立的请求，请求与请求之间并无关系。但在实际的应用场景，这种方式并不能满足我们的需求。举个大家都喜欢用的例子，把商品加入购物车，单独考虑这个请求，服务端并不知道这个商品是谁的，应该加入谁的购物车？因此这个请求的上下文环境实际上应该包含用户的相关信息，在每次用户发出请求时把这一小部分额外信息，也做为请求的一部分，这样服务端就可以根据上下文中的信息，针对具体的用户进行操作。所以这几种技术的出现都是对HTTP协议的一个补充，使得我们可以用HTTP协议+状态管理构建一个的面向用户的WEB应用。

Session 和Cookie的区别

这里我想先谈谈session与cookies,因为这两个技术是做为开发最为常见的。那么session与cookies的区别是什么？个人认为session与cookies最核心区别在于额外信息由谁来维护。利用cookies来实现会话管理时，用户的相关信息或者其他我们想要保持在每个请求中的信息，都是放在cookies中,而cookies是由客户端来保存，每当客户端发出新请求时，就会稍带上cookies,服务端会根据其中的信息进行操作。
当利用session来进行会话管理时，客户端实际上只存了一个由服务端发送的session_id,而由这个session_id,可以在服务端还原出所需要的所有状态信息，从这里可以看出这部分信息是由服务端来维护的。

除此以外，session与cookies都有一些自己的缺点：

cookies的安全性不好，攻击者可以通过获取本地cookies进行欺骗或者利用cookies进行CSRF攻击。使用cookies时,在多个域名下，会存在跨域问题。
session 在一定的时间里，需要存放在服务端，因此当拥有大量用户时，也会大幅度降低服务端的性能，当有多台机器时，如何共享session也会是一个问题.(redis集群)也就是说，用户第一个访问的时候是服务器A，而第二个请求被转发给了服务器B，那服务器B如何得知其状态。实际上，session与cookies是有联系的，比如我们可以把session_id存放在cookies中的。

JWT是如何工作的

首先用户发出登录请求，服务端根据用户的登录请求进行匹配，如果匹配成功，将相关的信息放入payload中，利用算法，加上服务端的密钥生成token，这里需要注意的是secret_key很重要，如果这个泄露的话，客户端就可以随机篡改发送的额外信息，它是信息完整性的保证。生成token后服务端将其返回给客户端，客户端可以在下次请求时，将token一起交给服务端，一般是说我们可以将其放在Authorization首部中，这样也就可以避免跨域问题。

146.简述Django请求生命周期

一般是用户通过浏览器向我们的服务器发起一个请求(request),这个请求会去访问视图函数，如果不涉及到数据调用，那么这个时候视图函数返回一个模板也就是一个网页给用户)
视图函数调用模型毛模型去数据库查找数据，然后逐级返回，视图函数把返回的数据填充到模板中空格中，最后返回网页给用户。

1.wsgi ,请求封装后交给web框架（Flask，Django)

2.中间件，对请求进行校验或在请求对象中添加其他相关数据，例如：csrf,request.session

3.路由匹配 根据浏览器发送的不同url去匹配不同的视图函数

4.视图函数，在视图函数中进行业务逻辑的处理，可能涉及到：orm，templates

5.中间件，对响应的数据进行处理

6.wsgi，将响应的内容发送给浏览器

147.用的restframework完成api发送时间时区

当前的问题是用django的rest framework模块做一个get请求的发送时间以及时区信息的api

```
class getCurrenttime(APIView):
    def get(self,request):
        local_time = time.localtime()
        time_zone =settings.TIME_ZONE
        temp = {'localtime':local_time,'timezone':time_zone}
        return Response(temp)
```

148.nginx,tomcat,apach到都是什么?

Nginx（engine x)是一个高性能的HTTP和反向代理服务器，也是 一个IMAP/POP3/SMTP服务器，工作在OSI七层，负载的实现方式：轮询，IP_HASH,fair,session_sticky.
Apache HTTP Server是一个模块化的服务器，源于NCSAhttpd服务器
Tomcat 服务器是一个免费的开放源代码的Web应用服务器，属于轻量级应用服务器，是开发和调试JSP程序的首选。

149.请给出你熟悉关系数据库范式有哪些，有什么作用?

在进行数据库的设计时，所遵循的一些规范，只要按照设计规范进行设计，就能设计出没有数据冗余和数据维护异常的数据库结构。

数据库的设计的规范有很多，通常来说我们在设是数据库时只要达到其中一些规范就可以了，这些规范又称之为数据库的三范式，一共有三条，也存在着其他范式，我们只要做到满足前三个范式的要求，就能设陈出符合我们的数据库了，我们也不能全部来按照范式的要求来做，还要考虑实际的业务使用情况，所以有时候也需要做一些违反范式的要求。
1.数据库设计的第一范式(最基本)，基本上所有数据库的范式都是符合第一范式的，符合第一范式的表具有以下几个特点：

数据库表中的所有字段都只具有单一属性，单一属性的列是由基本的数据类型（整型，浮点型，字符型等）所构成的设计出来的表都是简单的二比表

2.数据库设计的第二范式(是在第一范式的基础上设计的)，要求一个表中只具有一个业务主键，也就是说符合第二范式的表中不能存在非主键列对只对部分主键的依赖关系

3.数据库设计的第三范式，指每一个非主属性既不部分依赖与也不传递依赖于业务主键，也就是第二范式的基础上消除了非主属性对主键的传递依赖

150.简述QQ登陆过程

qq登录，在我们的项目中分为了三个接口，

第一个接口是请求qq服务器返回一个qq登录的界面;

第二个接口是通过扫码或账号登陆进行验证，qq服务器返回给浏览器一个code和state,利用这个code通过本地服务器去向qq服务器获取access_token覆返回给本地服务器，凭借access_token再向qq服务器获取用户的openid(openid用户的唯一标识)

第三个接口是判断用户是否是第一次qq登录，如果不是的话直接登录返回的jwt-token给用户，对没有绑定过本网站的用户，对openid进行加密生成token进行绑定

151.post 和 get的区别?

1.GET是从服务器上获取数据，POST是向服务器传送数据

2.在客户端，GET方式在通过URL提交数据，数据在URL中可以看到，POST方式，数据放置在HTML——HEADER内提交

3.对于GET方式，服务器端用Request.QueryString获取变量的值，对于POST方式，服务器端用Request.Form获取提交的数据

152.项目中日志的作用

一、日志相关概念

1.日志是一种可以追踪某些软件运行时所发生事件的方法

2.软件开发人员可以向他们的代码中调用日志记录相关的方法来表明发生了某些事情

3.一个事件可以用一个包含可选变量数据的消息来描述

4.此外，事件也有重要性的概念，这个重要性也可以被成为严重性级别(level)

二、日志的作用

1.通过log的分析，可以方便用户了解系统或软件、应用的运行情况;

2.如果你的应用log足够丰富，可以分析以往用户的操作行为、类型喜好，地域分布或其他更多信息;

3.如果一个应用的log同时也分了多个级别，那么可以很轻易地分析得到该应用的健康状况，及时发现问题并快速定位、解决问题，补救损失。

4.简单来讲就是我们通过记录和分析日志可以了解一个系统或软件程序运行情况是否正常，也可以在应用程序出现故障时快速定位问题。不仅在开发中，在运维中日志也很重要，日志的作用也可以简单。总结为以下几点：

1.程序调试

2.了解软件程序运行情况，是否正常

3,软件程序运行故障分析与问题定位

4,如果应用的日志信息足够详细和丰富，还可以用来做用户行为分析

153.django中间件的使用?

Django在中间件中预置了六个方法，这六个方法的区别在于不同的阶段执行，对输入或输出进行干预，方法如下：

1.初始化：无需任何参数，服务器响应第一个请求的时候调用一次，用于确定是否启用当前中间件

```
def __init__():
    pass
```

2.处理请求前：在每个请求上调用，返回None或HttpResponse对象。

```
def process_request(request):
    pass
```

3.处理视图前:在每个请求上调用，返回None或HttpResponse对象。

```
def process_view(request,view_func,view_args,view_kwargs):
    pass
```

4.处理模板响应前：在每个请求上调用，返回实现了render方法的响应对象。

```
def process_template_response(request,response):
    pass
```

5.处理响应后：所有响应返回浏览器之前被调用，在每个请求上调用，返回HttpResponse对象。

```
def process_response(request,response):
    pass
```

6.异常处理：当视图抛出异常时调用，在每个请求上调用，返回一个HttpResponse对象。

```
def process_exception(request,exception):
    pass
```

154.谈一下你对uWSGI和nginx的理解?

1.uWSGI是一个Web服务器，它实现了WSGI协议、uwsgi、http等协议。Nginx中HttpUwsgiModule的作用是与uWSGI服务器进行交换。WSGI是一种Web服务器网关接口。它是一个Web服务器（如nginx，uWSGI等服务器）与web应用（如用Flask框架写的程序）通信的一种规范。

要注意WSGI/uwsgi/uWSGI这三个概念的区分。

WSGI是一种通信协议。

uwsgi是一种线路协议而不是通信协议，在此常用于在uWSGI服务器与其他网络服务器的数据通信。

uWSGI是实现了uwsgi和WSGI两种协议的Web服务器。

nginx 是一个开源的高性能的HTTP服务器和反向代理：

1.作为web服务器，它处理静态文件和索引文件效果非常高

2.它的设计非常注重效率，最大支持5万个并发连接，但只占用很少的内存空间

3.稳定性高，配置简洁。

4.强大的反向代理和负载均衡功能，平衡集群中各个服务器的负载压力应用

155.Python中三大框架各自的应用场景?

django:主要是用来搞快速开发的，他的亮点就是快速开发，节约成本，,如果要实现高并发的话，就要对django进行二次开发，比如把整个笨重的框架给拆掉自己写socket实现http的通信,底层用纯c,c++写提升效率，ORM框架给干掉，自己编写封装与数据库交互的框架,ORM虽然面向对象来操作数据库，但是它的效率很低，使用外键来联系表与表之间的查询;
flask: 轻量级，主要是用来写接口的一个框架，实现前后端分离，提考开发效率，Flask本身相当于一个内核，其他几乎所有的功能都要用到扩展(邮件扩展Flask-Mail，用户认证Flask-Login),都需要用第三方的扩展来实现。比如可以用Flask-extension加入ORM、文件上传、身份验证等。Flask没有默认使用的数据库，你可以选择MySQL，也可以用NoSQL。

其WSGI工具箱用Werkzeug(路由模块)，模板引擎则使用Jinja2,这两个也是Flask框架的核心。

Tornado： Tornado是一种Web服务器软件的开源版本。Tornado和现在的主流Web服务器框架（包括大多数Python的框架）有着明显的区别：它是非阻塞式服务器，而且速度相当快。得利于其非阻塞的方式和对epoll的运用，Tornado每秒可以处理数以千计的连接因此Tornado是实时Web服务的一个理想框架

156.Django中哪里用到了线程？哪里用到了协程？哪里用到了进程?

1.Django中耗时的任务用一个进程或者线程来执行，比如发邮件，使用celery.

2.部署django项目是时候，配置文件中设置了进程和协程的相关配置。

157.有用过Django REST framework吗?

Django REST framework是一个强大而灵活的Web API工具。使用RESTframework的理由有：

Web browsable API对开发者有极大的好处

包括OAuth1a和OAuth2的认证策略

支持ORM和非ORM数据资源的序列化

全程自定义开发--如果不想使用更加强大的功能，可仅仅使用常规的function-based views额外的文档和强大的社区支持

158.对cookies与session的了解？他们能单独用吗?

Session采用的是在服务器端保持状态的方案，而Cookie采用的是在客户端保持状态的方案。但是禁用Cookie就不能得到Session。因为Session是用Session ID来确定当前对话所对应的服务器Session，而Session ID是通过Cookie来传递的，禁用Cookie相当于SessionID,也就得不到Session。

爬虫

159.试列出至少三种目前流行的大型数据库

160.列举您使用过的Python网络爬虫所用到的网络数据包?

requests, urllib,urllib2, httplib2

161.爬取数据后使用哪个数据库存储数据的，为什么?

162.你用过的爬虫框架或者模块有哪些？优缺点?

Python自带：urllib,urllib2

第三方：requests

框架： Scrapy

urllib 和urllib2模块都做与请求URL相关的操作，但他们提供不同的功能。

urllib2: urllib2.urlopen可以接受一个Request对象或者url,(在接受Request对象时，并以此可以来设置一个URL的headers),urllib.urlopen只接收一个url。

urllib 有urlencode,urllib2没有，因此总是urllib, urllib2常会一起使用的原因

scrapy是封装起来的框架，他包含了下载器，解析器，日志及异常处理，基于多线程，twisted的方式处理，对于固定单个网站的爬取开发，有优势，但是对于多网站爬取100个网站，并发及分布式处理不够灵活，不便调整与扩展

requests是一个HTTP库，它只是用来请求，它是一个强大的库，下载，解析全部自己处理，灵活性高

Scrapy优点：异步，xpath，强大的统计和log系统，支持不同url。shell方便独立调试。写middleware方便过滤。通过管道存入数据库

163.写爬虫是用多进程好？还是多线程好?

164.常见的反爬虫和应对方法?

165.解析网页的解析器使用最多的是哪几个?

166.需要登录的网页，如何解决同时限制ip，cookie,session

167.验证码的解决?

168.使用最多的数据库，对他们的理解?

169.编写过哪些爬虫中间件?

170.“极验”滑动验证码如何破解?

171.爬虫多久爬一次，爬下来的数据是怎么存储?

172.cookie过期的处理问题?

173.动态加载又对及时性要求很高怎么处理?

174.HTTPS有什么优点和缺点?

175.HTTPS是如何实现安全传输数据的?

176.TTL，MSL，RTT各是什么?

177.谈一谈你对Selenium和PhantomJS了解

178.平常怎么使用代理的？

179.存放在数据库(redis、mysql等)。

180.怎么监控爬虫的状态?

181.描述下scrapy框架运行的机制?

182.谈谈你对Scrapy的理解?

183.怎么样让 scrapy 框架发送一个 post 请求（具体写出来)

184.怎么监控爬虫的状态？

185.怎么判断网站是否更新?

186.图片、视频爬取怎么绕过防盗连接

187.你爬出来的数据量大概有多大？大概多长时间爬一次?

188.用什么数据库存爬下来的数据？部署是你做的吗？怎么部署?

189.增量爬取

190.爬取下来的数据如何去重，说一下scrapy的具体的算法依据。

191.Scrapy的优缺点?

192.怎么设置爬取深度?

193.scrapy和scrapy-redis有什么区别？为什么选择redis数据库?

194.分布式爬虫主要解决什么问题?

195.什么是分布式存储?

196.你所知道的分布式爬虫方案有哪些?

197.scrapy-redis，有做过其他的分布式爬虫吗?

数据库

MySQL

198.主键 超键 候选键 外键

主键：数据库表中对存储数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值(Null).

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

候选键：是最小超键，即没有冗余元素的超键。

外键：在一个表中存在的另一个表的主键称此表的外键。

199.视图的作用，视图可以更改么?

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询;不包含任何列或数据。使用视图可以简化复杂的sql操作，隐藏具体的细节，保护数据;视图创建后，可以使用与表相同的方式利用它们。

视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by则对视图再次order by将被覆盖。

创建视图： create view xxx as xxxxxx

对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新;但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

200.drop,delete与truncate的区别

drop直接删掉表，truncate删除表中数据，再插入时自增长id又从1开始，delete删除表中数据，可以加where字句。

1.delete 语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行回滚操作。truncate table则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器，执行速度快。

2.表和索引所占空间。当表被truncate后，这个表和索引所占用的空间会恢复到初始大小，而delete操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

3.一般而言，drop>truncate>delete

4.应用范围。truncate只能对table，delete可以是table和view

5.truncate和delete只删除数据，而drop则删除整个表（结构和数据)

6.truncate与不带where的delete:只删除数据，而不删除表的结构（定义）drop语句将删除表的结构被依赖的约束(constrain),触发器（trigger)索引(index);依赖于该表的存储过程/函数将被保留，但其状态会变为:invalid.

201.索引的工作原理及其种类

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询，更新数据库表中数据。索引的实现通常使用B树以其变种B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间（因为索引也要随之变动）

202.连接的种类

203.数据库优化的思路

204.存储过程与触发器的区别

205.悲观锁和乐观锁是什么?

206.你常用的mysql引擎有哪些?各引擎间有什么区别?

Redis

207.Redis宕机怎么解决?

宕机:服务器停止服务‘

如果只有一台redis，肯定 会造成数据丢失，无法挽救

多台redis或者是redis集群，宕机则需要分为在主从模式下区分来看：

slave从redis宕机，配置主从复制的时候才配置从的redis，从的会从主的redis中读取主的redis的操作日志1，在redis中从库重新启动后会自动加入到主从架构中，自动完成同步数据;

2, 如果从数据库实现了持久化，此时千万不要立马重启服务，否则可能会造成数据丢失，正确的操作如下：在slave数据上执行SLAVEOF ON ONE,来断开主从关系并把slave升级为主库，此时重新启动主数据库，执行SLAVEOF，把它设置为从库，连接到主的redis上面做主从复制，自动备份数据。

以上过程很容易配置错误，可以使用redis提供的哨兵机制来简化上面的操作。简单的方法:redis的哨兵(sentinel)的功能

208.redis和mecached的区别，以及使用场景

区别

1、redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可以用于缓存其他东西，例如图片，视频等等

2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list,set,hash等数据结构的存储

3、虚拟内存-redis当物流内存用完时，可以将一些很久没用的value交换到磁盘

4、过期策略-memcache在set时就指定，例如set key1 0 0 8，即永不过期。Redis可以通过例如expire设定，例如expire name 10

5、分布式-设定memcache集群，利用magent做一主多从，redis可以做一主多从。都可以一主一从

6、存储数据安全-memcache挂掉后，数据没了，redis可以定期保存到磁盘(持久化)

7、灾难恢复-memcache挂掉后，数据不可恢复，redis数据丢失后可以通过aof恢复

8、Redis支持数据的备份，即master-slave模式的数据备份

9、应用场景不一样，redis除了作为NoSQL数据库使用外，还能用做消息队列，数据堆栈和数据缓存等;Memcache适合于缓存SQL语句，数据集，用户临时性数据，延迟查询数据和session等

使用场景

1,如果有持久方面的需求或对数据类型和处理有要求的应该选择redis

2,如果简单的key/value存储应该选择memcached.

209.Redis集群方案该怎么做?都有哪些方案?

1,codis

目前用的最多的集群方案，基本和twemproxy一致的效果，但它支持在节点数量改变情况下，旧节点数据客恢复到新hash节点

2redis cluster3.0自带的集群，特点在于他的分布式算法不是一致性hash，而是hash槽的概念，以及自身支持节点设置从节点。具体看官方介绍

3.在业务代码层实现，起几个毫无关联的redis实例，在代码层，对key进行hash计算，然后去对应的redis实例操作数据。这种方式对hash层代码要求比较高，考虑部分包括，节点失效后的替代算法方案，数据震荡后的字典脚本恢复，实例的监控，等等

210.Redis回收进程是如何工作的

一个客户端运行了新的命令，添加了新的数据。

redis检查内存使用情况，如果大于maxmemory的限制，则根据设定好的策略进行回收。

一个新的命令被执行等等，所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断回收回到边界以下。

如果一个命令的结果导致大量内存被使用(例如很大的集合的交集保存到一个新的键)，不用多久内存限制就会被这个内存使用量超越。

MongoDB

211.MongoDB中对多条记录做更新操作命令是什么?

212.MongoDB如何才会拓展到多个shard里?

测试

213.编写测试计划的目的是

214.对关键词触发模块进行测试

215.其他常用笔试题目网址汇总

216.测试人员在软件开发过程中的任务是什么

217.一条软件Bug记录都包含了哪些内容?

218.简述黑盒测试和白盒测试的优缺点

219.请列出你所知道的软件测试种类，至少5项

220.Alpha测试与Beta测试的区别是什么?

221.举例说明什么是Bug？一个bug report应包含什么关键字?

数据结构

222.数组中出现次数超过一半的数字-Python版

223.求100以内的质数

224.无重复字符的最长子串-Python实现

225.通过2个5/6升得水壶从池塘得到3升水

226.什么是MD5加密，有什么特点?

227.什么是对称加密和非对称加密

228.冒泡排序的思想?

229.快速排序的思想?

230.如何判断单向链表中是否有环?

231.你知道哪些排序算法（一般是通过问题考算法)

232.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]
```

233.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
```

234.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
```

235.两数之和 Two Sum

236.搜索旋转排序数组 Search in Rotated Sorted Array

237.Python实现一个Stack的数据结构

238.写一个二分查找

239.set 用 in 时间复杂度是多少，为什么?

240.列表中有n个正整数范围在[0，1000]，进行排序;

241.面向对象编程中有组合和继承的方法实现新的类

大数据

242.找出1G的文件中高频词

243.一个大约有一万行的文本文件统计高频词

244.怎么在海量数据中找出重复次数最多的一个?

245.判断数据是否在大量数据中

架构

Python后端架构演进

这篇文章几乎涵盖了python会用的架构，在面试可以手画架构图，根据自己的项目谈下技术选型和优劣，遇到的坑等。绝对加分