Python算法岗题目整理

1. 一行代码实现1-100的和

sum(range(100))

1. 如何实现在函数内部修改全局变量？

在修改前使用global关键字先声明

global v

v=20

1. 列出5个python标准库

os, glob, sys, math, random, datetime

1. 字典如何删除键和合并两个字典

del删除键，使用{\*\*x, \*\*y}进行合并

补充python中\*和\*\*的四种用途 运算符、函数形参、函数实参（解引用）、序列解包（只有\*）

1. 谈下python的GIL

GIL是CPython解释器中的全局解释锁，是解释器的特性而非语言特性。GIL是python语言遗留下来的一个bug。在现代多核处理器中，同一进程下的子进程共享同一把GIL，线程占据GIL之后才能获得CPU执行权。这就导致在多核处理器上一个时刻只能有一个核心执行线程，其他核心资源被浪费。

在IO密集型任务中，比如web服务，GIL的影响不是很明显，因为当线程出现IO操作时会主动让出CPU，而由于IO等待视唱是不确定的，因此其他线程一定能获得CPU执行权。但在计算密集型任务中，线程在时间片结束时候，会再次参与到CPU执行权的竞争中，很有可能再次抢占CPU。因此计算密集型任务中，GIL的存在一方面使得多核资源未被充分利用，另一方面也可能使得线程获得CPU的执行时间不够公平。因此计算密集型任务中一般使用多进程，或者使用C代码绕过GIL。

1. Python list去重方法

list(set(L))

1. fun(\*args, \*\*kwargs)

\*和\*\*的函数形参用法，用于绑定位置参数和关键词参数

1. python2和python3中的range(100)区别

Python3中range使用迭代器实现（实现了\_\_iter\_\_方法的对象成为迭代器，如果实现了\_\_next\_\_方法便能遍历）

Python2中使用列表实现

1. \_\_new\_\_和\_\_init\_\_区别

\_\_new\_\_是静态方法，\_\_init\_\_是实例方法

\_\_new\_\_返回创建的实例，\_\_init\_\_什么都不返回

一般\_\_new\_\_先被调用得到实例，之后使用\_\_init\_\_初始化实例

使用\_\_new\_\_关键字可用来实现单例模式

1. with方法打开文件帮我们做了什么？

首先with完成了f.open和f.close操作，其次在打开文件发生异常时，with能够自动帮我们实现一套try except和finally的逻辑，确保异常得到处理，同时文件被关闭。

1. 列表[1,2,3,4,5],请使用map()函数输出[1,4,9,16,25]，并使用列表推导式提取出大于10的数，最终输出[16,25]

l2=list(map(lambda x: x\*\*2, l1))

l2=[x for x in l2 if x>10]

1. Python2和python3的区别

我现在不用python2了

1. 列出python中可变数据类型和不可变数据类型

可变数据类型：list（链表实现）和dict（hash map实现）

不可变数据类型：数值型（int, float），字符串型（字符数组），元组（不实现运算符重载，导致不能被修改）

补充hashmap原理：hashmap主干采用数组，关键字通过hash函数计算得到存储下标。当出现下标冲突时，一般使用链地址法解决冲突。冲突元素的地址均挂在hash主干的同一个节点上。这样，不冲突时主干节点下面挂的第一个地址就是要搜索的元素，出现冲突时则比较该子链上的关键字。Hashmap是典型的空间换时间，一般查找、修改hashmap时间复杂度是O(1)。hash函数的设计对于hashmap的性能影响比较重要。

1. s = "ajldjlajfdljfddd"，去重并从小到大排序输出

''.join(sorted(list(set(list(s)))))

1. 字典按照键从小到大排序

{e[0]:e[1] for e in sorted(x.items(), key=lambda x:x[0])}

1. 利用collections的Counter方法统计字符串每个单词出现的次数

from collections import Counter

a = "kjalfj;ldsjafl;hdsllfdhg;lahfbl;hl;ahlf;h"

print(Counter(a))

1. 利用filter方法求出所有奇数并构造新列表a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

b = list(filter(lambda x: x%2==1, a))

1. a=（1，）b=(1)，c=("1") 分别是什么类型的数据

a = (1,)元组，b=(1) int c=(“1”)str

1. 举例说明异常模块中try except else finally的相关意义

try..except..else先执行try，捕获到异常，执行except，没有捕获到异常，执行else语句

try..except..finally先执行try，捕获到异常，执行except，最后一定执行finally

1. 举例说明zip（）函数用法

zip()函数在运算时，会以一个或多个可迭代对象做为参数，返回一个元组的列表,同时将这些序列中并排的元素配对。当传入参数的长度不同时，zip能自动以最短序列长度为准进行截取，获得元组。

1. 提高python运行效率的方法

使用生成器，因为可以节约大量内存

循环代码优化，避免过多重复代码的执行

核心模块用Cython编译为C动态库，提高效率

多进程、多线程

多个if elif条件判断，可以把最有可能先发生的条件放到前面写，这样可以减少程序判断的次数，提高效率

1. python中copy和deepcopy区别

**复制不可变数据类型**，不管copy还是deepcopy,都是浅复制，和=“赋值”的情况一样

**复制的值是可变对象**（列表和字典）

copy有两种情况：

第一种情况：复制的对象中无复杂子对象，原来值的改变并不会影响浅复制的值，同时浅复制的值改变也并不会影响原来的值。原来值的id值与浅复制原来的值不同。

第二种情况：复制的对象中有复杂子对象 （例如列表中的一个子元素是一个列表）， 改变原来的值 中的复杂子对象的值，会影响浅复制的值。

deepcopy：完全复制独立，包括内层列表和字典

1. 列举python魔术方法

\_\_new\_\_构建实例

\_\_init\_\_实例初始化

\_\_del\_\_析构函数，释放资源

\_\_class\_\_获得已知类的对象

\_\_str\_\_实例的字符串描述（print调用）

\_\_repr\_\_功能类似上述，给开发人员看，直接交互式环境输出实例名字

1. 请将[i for i in range(3)]改成生成器

生成器和迭代器的关系：生成器是一种特殊的迭代器，生成器语法更简单，功能更强大点，性能和迭代器一样

def g(n):

i=0

while i<n:

yield i

i+=1

[i for i in g(10)]

1. 使用lambda函数对list排序foo = [-5,8,0,4,9,-4,-20,-2,8,2,-4]，输出结果为[0,2,4,8,8,9,-2,-4,-4,-5,-20]，正数从小到大，负数从大到小

Sorted(foo, key=lambda x: (x<0, abs(x)))

1. 两种方法去空格

str.replace(‘ ‘, ‘’)

‘’.join(str.split(‘ ‘))

1. python传参数是传值还是传址?

Python是不允许程序员选择采用传值还是传址的。Python参数传递采用的是“**传对象引用**”的方式，这种方式相当于传值和传址的一种综合。如果函数收到的是一个可变对象(比如字典或者列表)的引用，就能修改对象的原始值——相当于传址。如果函数收到的是一个不可变对象(比如数字、字符或者元组)的引用，就不能直接修改原始对象——相当于传值。

1. 求两个列表的交集，并集，差集

a = [1, 2, 3, 4]

b = [4, 3, 5, 6]

set(a).intersection(set(b))

set(a).union(set(b))

set(a).difference(set(b))

set(b).difference(set(a))

1. 迭代器和生成器

str、list、set、dict是可迭代的，但是不是迭代器

迭代器持有一个内部状态的字段，用于记录下次迭代返回值，它实现了\_\_next\_\_和\_\_iter\_\_方法，迭代器不会一次性把所有元素加载到内存，而是需要的时候才生成返回结果

生成器是一种特殊的迭代器，它的返回值不是通过return而是用yield

每个生成器都是一个迭代器，但是反过来不行。通常生成器是通过调用一个或多个yield表达式构成的函数s生成的。同时满足迭代器的定义。当你需要一个类除了有生成器的特性之外还要有一些自定义的方法时，可以使用自定义的迭代器，一般来说生成器更方便，更简单。

1. @staticmethod和@classmethod

Python有三个方法，静态方法，类方法，实例方法（没有任何装饰器）。

class A(object):

def foo(self, x):

print(“executing foo”)

@classmethod

def class\_foo(cls, x):

print(“excuting class\_foo”)

@staticmethod

def static\_foo(x):

print(“excuting static\_foo”);

self,cls是对类或者实例的绑定，类方法和实例方法在定义时都要对用这两个关键字进行绑定，而静态方法不需要对谁进行绑定。静态方法和类方法都可以用类或者实例进行调用。

什么是python装饰器？

装饰器(Decorators)有助于让我们的代码更简短，将核心函数包装在内部，外围做些修饰性的工作，如打印日志。

1. 类变量和实例变量

类变量是在类的所有实例之间共享的值，定义在类的开头，不在任何函数内部

实例变量隶属每个实例，在类函数内部创建，使用self.x=的形式

1. Python自省机制

运行时获得对象类型，type, dir, isinstance, hasattr

dir: 返回传递给它的任何对象的属性名称经过排序的列表。如果不指定对象，则 dir() 返回当前作用域中的名称;

type: 返回对象类型

isinstance: 确定它是否是某个特定类型或定制类的实例 isintance(x, (list, int)) isinstance(x, int)

hasattr: 测试一个或多个属性是否存在，长和getattr一起使用

1. Python中单下划线和双下划线

单下划线：只是表示类成员变量或者成员函数是私有的，但不是真正的私有，还是能直接被访问；

双下划线：类成员前面添加了双下划綫后，在类外调用需要采用 实例.\_类名\_\_foo的方式来访问实例的\_\_foo成员函数。这样设计的目的是为了继承时避免父类成员函数被子类覆盖；

前后都有双下划线：python自定义的一些内部函数

1. Python中的重载

Python中不存在函数重载，也不需要函数重载。函数重载是形参类型不同或者形参数量不停。Python中函数可以传递任意类型参数，因此第一点满足需求；同时参数数量的变化可用缺省参数来实现，第二点也能满足需求。因此python中不存在函数重载操作。

1. 什么是协程？

协程是轻量级线程，协程基于线程实现。协程相比线程主要有两个特点，协程的栈空间比线程小很多，二是协程之间的切换在用户态进行，无需操作系统调度。因此，协程一般是基于线程库在用户层面实现。

1. Python的垃圾回收机制
2. 引用计数

每个对象维护一个ob\_refcnt成员变量，创建对象，对象被引用，对象被传入函数，对象作为一个元素存储在容器中会让引用技术+1，显示del对象，引用别名被赋值新对象，离开函数作用于，从容器中删除或者容器被销毁会导致引用计数-1。

引用计数变为0时，对象生命就结束了。

优点：简单，实时性

缺点：维护引用计数消耗资源（需要不断修改引用计数值）

循环引用情况下资源没法得到正确释放（和C++的智能指针share\_ptr类似）

为了改进上述问题，引用了标记-清除机制

1. 标记—清除机制

为了解决孤岛状态的对象（只有内部引用，没有外部引用），必须要能区分内部引用和外部引用，当只有外部引用时，需要进行垃圾回收。将对象的引用（指针）关系构建成有向图，从根对象出发，能够到达的是活动对象，不可达的是非活动对象。对于图中的环（即循环引用），根据规则减掉引用计数（相当于减少了边）。标记-清除技术主要是用于容器对象，对于数值型，字符串型不存在循环引用问题。

但是如果全部的对象引用都放在一个图中，那么每次执行标记-清除就需要遍历整个栈内存对象，十分耗时，并且何时执行这个机制也是一个问题。因此采用了分代回收。

1. 分代回收

开始创建对象时，所有的对象全部放入一个称作零代链表的有向图中。

统计分配计数值和释放计数值，当两个值之间的差距超过阈值时，启动标记-清除机制，非活动对象被清除，活动对象则被转移到一代链表中。通过设置阈值可以控制活跃对象在零代，一代，二代之间的转移。

1. 单例模式

确保一个类只有一个实例存在。比如某个服务器程序的配置信息存放在一个文件中，客户端通过一个Config 的类来读取配置文件的信息。如果在程序运行期间，有很多地方都需要使用配置文件的内容，也就是说，很多地方都需要创建Config 对象的实例，这就导致系统中存在多个Config 的实例对象，严重浪费内存资源。类似Config 这样的类，我们希望在程序运行期间只存在一个实例对象。

Python实现单例模式的方法：

**使用模块**：将需要实现的单例功能放到一个py文件中，并且在该py文件中构建一个实例，在其他文件中import该实例即可。实现原理：Python 的模块就是天然的单例模式，因为模块在第一次导入时，会生成 .pyc 文件，当第二次导入时，就会直接加载 .pyc 文件，而不会再次执行模块代码。因此，我们只需把相关的函数和数据定义在一个模块中，就可以获得一个单例对象了。

class Singleton(object):

def foo(self):

pass

singleton = Singleton()

**使用装饰器:**

def Singleton(cls):

\_instance = {}

def \_singleton (\*args, \*\*kargs):

if cls not in \_instance:

\_instance[cls] = cls(\*args, \*\*kargs)

return \_instance[cls]

return \_singleton

@Singleton

class A(object):

a = 1

def \_\_init\_\_ (self, x=0):

self.x = x

a1 = A(2)

a2 = A(3)

上述代码中a2.x=a1.x=2，是同一个对象不会去创建新的对象。

**使用\_\_new\_\_**：

class Musicplayer:

instance=None

def \_\_new\_\_ (cls, \*args, \*\*kwargs):

if cls.instance is None:

cls.instance=super().\_\_new\_\_(cls)

return cls.instance

**使用metaclass：**元类，不是怎么了解

38.\_\_all\_\_使用

\_\_all\_\_放在module.py文件中，只影响from module import \*包含的内容，不影响from module import xxx

\_\_all\_\_放在包package的\_\_init\_\_.py文件中，package中包含module.py，放在\_\_all\_\_的东西，可以直接从package中引入，而不需要使用package.module。