1 Python是如何进行内存管理的?

从三个方面来说,一对象的引用计数机制,二垃圾回收机制,三内存池机制

1.1 对象的引用计数机制

python内部使用引用计数，来保持追踪内存中的对象，所有对象都有引用计数。

引用计数增加的情况：

1，一个对象分配一个新名称

2，将其放入一个容器中（如列表、元组或字典）

引用计数减少的情况：

1，使用del语句对对象别名显示的销毁

2，引用超出作用域或被重新赋值

sys.getrefcount( )函数可以获得对象的当前引用计数

多数情况下，引用计数比你猜测得要大得多。对于不可变数据（如数字和字符串），解释器会在程序的不同部分共享内存，以便节约内存。

1.2 垃圾回收

1，当一个对象的引用计数归零时，它将被垃圾收集机制处理掉。

2，当两个对象a和b相互引用时，del语句可以减少a和b的引用计数，并销毁用于引用底层对象的名称。然而由于每个对象都包含一个对其他对象的应用，因此引用计数不会归零，对象也不会销毁。（从而导致内存泄露）。为解决这一问题，解释器会定期执行一个循环检测器，搜索不可访问对象的循环并删除它们。

1.3 内存池机制

Python提供了对内存的垃圾收集机制，但是它将不用的内存放到内存池而不是返回给操作系统。

1，Pymalloc机制。为了加速Python的执行效率，Python引入了一个内存池机制，用于管理对小块内存的申请和释放。

2，Python中所有小于256个字节的对象都使用pymalloc实现的分配器，而大的对象则使用系统的malloc。

3，对于Python对象，如整数，浮点数和List，都有其独立的私有内存池，对象间不共享他们的内存池。也就是说如果你分配又释放了大量的整数，用于缓存这些整数的内存就不能再分配给浮点数。

2 Python垃圾回收机制

Python GC主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率。

1 引用计数

PyObject是每个对象必有的内容，其中ob_refcnt就是做为引用计数。当一个对象有新的引用时，它的ob_refcnt就会增加，当引用它的对象被删除，它的ob_refcnt就会减少.引用计数为0时，该对象生命就结束了。

优点:

简单
实时性
缺点:

维护引用计数消耗资源

循环引用

2 标记-清除机制

基本思路是先按需分配，等到没有空闲内存的时候从寄存器和程序栈上的引用出发，遍历以对象为节点、以引用为边构成的图，把所有可以访问到的对象打上标记，然后清扫一遍内存空间，把所有没标记的对象释放。

3 分代技术

分代回收的整体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每个集合就成为一个“代”，垃圾收集频率随着“代”的存活时间的增大而减小，存活时间通常利用经过几次垃圾回收来度量。

Python默认定义了三代对象集合，索引数越大，对象存活时间越长。

举例： 当某些内存块M经过了3次垃圾收集的清洗之后还存活时，我们就将内存块M划到一个集合A中去，而新分配的内存都划分到集合B中去。当垃圾收集开始工作时，大多数情况都只对集合B进行垃圾回收，而对集合A进行垃圾回收要隔相当长一段时间后才进行，这就使得垃圾收集机制需要处理的内存少了，效率自然就提高了。在这个过程中，集合B中的某些内存块由于存活时间长而会被转移到集合A中，当然，集合A中实际上也存在一些垃圾，这些垃圾的回收会因为这种分代的机制而被延迟。

值和引用

在Python中，数字、字符或者元组等不可变对象类型都属于值传递，而字典dict或者列表list等可变对象类型属于引用传递。

如果要想修改新赋值后原对象不变，则需要用到python的copy模块，即对象拷贝。

copy.copy属于浅拷贝，拷贝的是第一层list，而copy.deepcopy属于深拷贝，对list所有子元素都进行深拷贝。

python实现单例

装饰器

装饰器本质上是一个Python函数，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。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、权限校验等场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

简单装饰器

```
def use_logging(func):

    def wrapper(*args, **kwargs):
        logging.warn("%s is running" % func.__name__)
        return func(*args, **kwargs)
    return wrapper

def bar():
    print('i am bar')

bar = use_logging(bar)
bar()
```

函数use_logging就是装饰器，它把执行真正业务方法的func包裹在函数里面，看起来像bar被use_logging装饰了。在这个例子中，函数进入和退出时 ，被称为一个横切面(Aspect)，这种编程方式被称为面向切面的编程(Aspect-Oriented Programming)。

python 装饰器实现

- 使用函数装饰器实现单例
- 使用类装饰器实现单例
- 使用 **new** 关键字实现单例
- 使用 metaclass 实现单例

装饰器：参考 https://www.zhihu.com/question/26930016

使用函数装饰器实现单例

通过id来看引用a的内存地址可以比较理解：

```
def singleton(cls):
    _instance = {}

    def inner():
        if cls not in _instance:
            _instance[cls] = cls()
        return _instance[cls]
    return inner

@singleton
class Cls(object):
    def __init__(self):
        pass

cls1 = Cls()
cls2 = Cls()
print(id(cls1) == id(cls2))
```

类装饰器

```
class Singleton(object):
    def __init__(self, cls):
        self._cls = cls
        self._instance = {}
    def __call__(self):
        if self._cls not in self._instance:
            self._instance[self._cls] = self._cls()
        return self._instance[self._cls]

@Singleton
class Cls2(object):
    def __init__(self):
        pass

cls1 = Cls2()
cls2 = Cls2()
print(id(cls1) == id(cls2))
```

元类实现

```
def func(self):
    print("do sth")

Klass = type("Klass", (), {"func": func})

c = Klass()
c.func()
```

以上，我们使用 type 创造了一个类出来。这里的知识是 mataclass 实现单例的基础。

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class Cls4(metaclass=Singleton):
    pass

cls1 = Cls4()
cls2 = Cls4()
print(id(cls1) == id(cls2))
```

设计模式python实现

python 23种常用模式设计总结

交集 差集

交集： [val for val in b1 if val in b2]

差集： [val for val in b1 if val not in b2]

Python自省

这个也是python彪悍的特性.

自省就是面向对象的语言所写的程序在运行时,所能知道对象的类型.简单一句就是运行时能够获得对象的类型.比如type(),dir(),getattr(),hasattr(),isinstance().

```
a = [1,2,3]
b = {'a':1,'b':2,'c':3}
c = True
print type(a),type(b),type(c) # <type 'list'> <type 'dict'> <type 'bool'>
print isinstance(a,list)  # True
```

推导式

字典推导式：

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

列表生成式：

> > > L = [x*x for x in range(10)]
> > > L
> > > [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

python中is和==的区别

Python中对象包含的三个基本要素，分别是：id(身份标识) 、type(数据类型)和value(值)。

‘==’比较的是value值， ‘is’比较的是id

简述read、readline、readlines的区别：read读取整个文件，readline读取下一行数据，readlines读取整个文件到一个迭代器以供我们遍历（读取到一个list中，以供使用，比较方便）

*args and **kwargs

当你不确定你的函数里将要传递多少参数时你可以用*args.例如,它可以传递任意数量的参数:

**kwargs允许你使用没有事先定义的参数名

Python 不支持函数重载

函数重载主要是为了解决两个问题。

可变参数类型。

可变参数个数。

python可以设置缺省参数，解决问题2。缺省参数。对那些缺少的参数设定为缺省参数。

Python中的作用域

当 Python 遇到一个变量的话他会按照这样的顺序进行搜索：

本地作用域（Local）→当前作用域被嵌入的本地作用域（Enclosing locals）→全局/模块作用域（Global）→内置作用域（Built-in）

GIL线程全局锁

线程全局锁(Global Interpreter Lock),即Python为了保证线程安全而采取的独立线程运行的限制,说白了就是一个核只能在同一时间运行一个线程.对于io密集型任务，python的多线程起到作用，但对于cpu密集型任务，python的多线程几乎占不到任何优势，还有可能因为争夺资源而变慢。

解决办法就是多进程和下面的协程(协程也只是单CPU,但是能减小切换代价提升性能).

协程 -- yield

简单点说协程是进程和线程的升级版,进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间,而协程就是用户自己控制切换的时机,不再需要陷入系统的内核态.

传统的生产者-消费者模型是一个线程写消息，一个线程取消息，通过锁机制控制队列和等待，但一不小心就可能死锁。

如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换回生产者继续生产，效率极高：

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

```
def fab(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b      # 使用 yield
        # print b
        a, b = b, a + b
        n = n + 1

for n in fab(5):
    print n
```

yield的作用就是把一个函数变成一个 generator,调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 fab 函数内部的代码，执行到 yield b 时，

fab 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。

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

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

调用父类super() 函数

super() 函数的一个常见用法是在 **init**() 方法中确保父类被正确的初始化了：

super() 的另外一个常见用法出现在覆盖Python特殊方法的代码中，比如：

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value) # Call original __setattr__
        else:
            setattr(self._obj, name, value)
```

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'))
```

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。

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

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

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。

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。

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.

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

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
```

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。
'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]])

e、使用re模块进行查找和替换：

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')

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

简述with方法打开处理文件帮我我们做了什么?

打开文件在进行读写的时候可能会出现一些异常状况，如果按照常规的f.open

写法，我们需要try,except,finally，做异常判断，并且文件最终不管遇到什么情况，都要执行finally f.close()关闭文件，with方法帮我们实现了finally中f.close

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。

```
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

```
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。

```
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()
```

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。

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。

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

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。

举例说明zip（）函数用法

zip()函数在运算时，会以一个或多个序列（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。

zip()参数可以接受任何类型的序列，同时也可以有两个以上的参数;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。

list tuple区别

相同点:

在 Python 中都是序列类型的容器对象；

可以支持任何类型的数据存放；

同样支持切片和迭代等操作；

元祖可以看做是一个不可变的列表。

不同点:

列表是可变的，元祖是不可变；我们可以修改列表的值，但是无法修改元祖的值；列表长度大小不固定，元祖长度大小固定。

因为刚刚说的元祖不可变，长度固定，因此在列表当中可以使用的增删改方法函数在元祖当中均没有。

元祖和列表的拷贝，并不会开辟新的地址存放元祖，但是会单独开辟一个新的空间存放列表，因为元组的不可变性，所以元组无法复制。

可以使用元组作为字典的键，但是列表不可以！元祖可以存放在集合当中，但是列表不可以！Python调用函数返回多个值的时候，使用元祖来存储其中的多个值（拆包）。