目录

[**零碎的问题** 3](#_Toc49948774)

[**内存管理和垃圾回收** 3](#_Toc49948775)

[**内存池机制** 3](#_Toc49948776)

[**垃圾回收机制** 3](#_Toc49948777)

[**python中有垃圾回收机制，在以下情况下仍然会有内存泄漏的问题：** 4](#_Toc49948778)

[**GIL（GlobalInterpreterLock）** 4](#_Toc49948779)

[**什么是协程** 5](#_Toc49948780)

[**list、str、dict、set实现细节和区别** 5](#_Toc49948781)

[**list** 5](#_Toc49948782)

[**tuple** 5](#_Toc49948783)

[**dict:** 5](#_Toc49948784)

[**set** 5](#_Toc49948785)

[**frozenset** 5](#_Toc49948786)

[**为什么list不能当dict的key** 5](#_Toc49948787)

[**tuple是不是冗余设计** 6](#_Toc49948788)

[**可迭代对象collections.abc.Iterable、迭代器Iterator、生成器Generator** 6](#_Toc49948789)

[**哪些是可迭代对象** 6](#_Toc49948790)

[**为什么要有迭代器：** 6](#_Toc49948791)

[**关于for循环** 6](#_Toc49948792)

[**生成器既是可迭代对象也是迭代器** 6](#_Toc49948793)

[**内存溢出和内存泄漏的区别** 6](#_Toc49948794)

[**直接赋值、深拷贝和浅拷贝** 7](#_Toc49948795)

[**可变类型和不可变类型** 7](#_Toc49948796)

[**解释型语言和编译型语言的区别** 7](#_Toc49948797)

[**为什么说python比较慢？** 7](#_Toc49948798)

[**‘\_\_new\_\_’与‘\_\_init\_\_’的区别** 7](#_Toc49948799)

[**@classmethod和@staticmethod的区别，以及分别运用在哪些使用场景？** 7](#_Toc49948800)

[**切片** 8](#_Toc49948801)

[**python2和python3的区别** 8](#_Toc49948802)

[**pythonv.s.java** 8](#_Toc49948803)

[**高阶函数** 8](#_Toc49948804)

[**is与==的区别** 9](#_Toc49948805)

[**pass** 9](#_Toc49948806)

[**python中的参数类型** 9](#_Toc49948807)

[**单下划线和双下划线的区别** 9](#_Toc49948808)

[**重写和重载的区别** 9](#_Toc49948809)

[上下文管理器 9](#_Toc49948810)

[**强类型、弱类型、动态语言、静态语言** 10](#_Toc49948811)

[**鸭子类型** 10](#_Toc49948812)

[**元类** 11](#_Toc49948813)

[**python多继承的顺序是什么** 11](#_Toc49948814)

[**多线程的实现方式** 11](#_Toc49948815)

[**多线程的同步** 12](#_Toc49948816)

[**多线程信息通信机制** 12](#_Toc49948817)

[**想要在当前线程中创建一个全局变量，并且各自线程互不干扰？** 13](#_Toc49948818)

[**线程池** 13](#_Toc49948819)

[**应用场景：** 13](#_Toc49948820)

[**守护线程** 14](#_Toc49948821)

[**运行一个py文件,分为三个步骤** 14](#_Toc49948822)

[**同步异步-阻塞非阻塞** 14](#_Toc49948823)

**零碎的问题**

**内存管理和垃圾回收**

**内存池机制**

预先在内存中申请一定数量的，大小相等的内存块留作备用，当有新的内存需求时，就先从内存池中分配内存给这个需求，不够了之后再申请新的内存。这样做最显著的优势就是能够减少内存碎片，提升效率。

**垃圾回收机制**

内存池机制具有内存的创建那就需要垃圾回收。**python采用了引用计数为主，标志清除和分代回收为辅测策略。**

好好计划

**引用计数**：每一个对象，在源码里就是一个结构体表示，都会有一个计数字段。当一个对象有新的引用时，它的引用计数就会增加，当引用它的对象被删除，它的引用计数就会减少。对象的引用计数为0，该对象不会立即被回收。

源码中维护了一个static双向环状链表refchain，所有对象都会放在refchain中，创建对象后会把该对象加入到refchain中，对象引用计数为0时，则将该对象从refchain中删除。

涉及两个结构体：

PyObject：指向上一个对象的指针、指向下一个对象的指针、类型、引用计数

PyVarObject：PyObject、元素个数

基于这两个结构体，每种类型的对象都会进行封装，

e.g.float🡪PyFloatObject：PyObject、double

e.g.list🡪PyListObject：PyVarObject、PyObject\*\*、allocated??

例如：a=9999b=a只创建了一个对象，该对象的值为9999，引用计数为2

优点:1)简单2)实时:一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。

缺点:1)不能解决对象的循环引用2)需要额外的空间维护引用计数。

**标记清除：**

为解决循环引用问题，在内存中额外维护一个链表L，专门存放可能存在循环引用的对象。

1. 标记阶段，gc会把所有的活动对象打上标记。

在某种情况下触发回收机制，会扫描L中的每一个元素，检查是否有循环引用，若有则循环引用中的对象的引用计数器减1。

对象之间通过引用（指针）连在一起，构成一个有向图，对象构成这个有向图的节点，而引用关系构成这个有向图的边。从根对象出发，沿着有向边遍历对象，可达的对象标记为活动对象，不可达的对象就是要被清除的非活动对象。**根对象就是全局变量、调用栈、寄存器。**

1. 清除阶段把没有标记的对象，视作非活动对象，进行回收。

优点：主要解决循环引用问题。主要处理的是容器对象，e.g.list、dict、tuple，因为数值、字符串对象不可能有循环引用问题。

缺点:清楚非活动对象前，必须顺序扫描整个堆内存，哪怕只剩下小部分活动对象，也要扫描所有对象。

**分代回收**：一种空间换时间的方式。根据对象的存活时间分为不同的集合，每个集合称为一个代，python分了三代，年轻代，中年代，老年代，对应三个链表，每个链表垃圾收集的频率随着对象存活时间的增大而减小。新创建的对象都会分配在年轻代，年轻代链表的总数达到上限时，Python垃圾收集机制就会被触发，把那些可以被回收的对象回收掉，而那些不会回收的对象就会被移到中年代去，依此类推，老年代中的对象是存活时间最久的对象，甚至是存活于整个系统的生命周期内。同时，分代回收是建立在标记清除技术基础之上。也是处理的容器对象。

把L分成三个链表：

0代：对象个数达700个，进行垃圾回收。

1代：0代垃圾回收10次，进行垃圾回收。

2代：1代垃圾回收10次，进行垃圾回收。

**python的缓存优化**

1. 池，为了避免重复创建和销毁对象，python维护了池。
   1. int类型，启动解释器是，会自动在缓存中生成[-5,256]的数字，这样子,a=1就不需要创建对象，直接在缓存中取即可；
   2. str类型，维护一个字符池，即把所有ascii字符缓存起来。

p.s.对于只包含字母、数字、下划线的字符串，python会在内部做驻留机制，即若内存中已存在某字符串，则不会重新创建，而是使用原来的地址。

1. free\_list，对象引用计数为0，在内部不会直接回收，而是将其添加至对应的free\_list链表中缓存，以后再创建对象，不用重新开辟内存，直接使用free\_list中的内存，若free\_list已满（M），才会销毁对象，释放内存。
   1. float，M=100。list，M=80。dict，M=80。
   2. tuple，M=20。tuple比较特殊，free\_list[0]专门存放空元组，free\_list[i]专门存放有i个元素的元组。

**python中有垃圾回收机制，在以下情况下仍然会有内存泄漏的问题：**

1）对象一直被全局变量所引用,全局变量生命周期长.

2）垃圾回收机被禁用或者设置成debug状态,垃圾回收的内存不会被释放.

3）循环引用过多,gc无法及时释放。

**GIL（GlobalInterpreterLock）**

python使用引用计数，在多线程下会出现引用计数问题，即一个对象的引用计数问题如果数据被错误地修改，很可能会导致一个对象一直不被回收，或者回收了一个不能被回收的对象。而如果给每个对象都加一把锁，每次访问加锁、解锁的开销太大，而且很容易带来死锁，因此，python加了一把超级大锁，引入了GIL。

Python有一个GIL（GlobalInterpreterLock）机制(有了这样一个设定，就不用管线程之间的加锁、同步等问题了)，任何线程在运行之前必须获取这个全局锁才能执行，每当执行完100个tricks(一个trick对应1条或多条字节码)，全局锁才会释放，切换到其他线程执行。在pythonGIL之下，同一时刻只能有一个线程在运行，python只能使用到1个核。

而且，还会有一个问题，假设有线程A,B,C，比如当前执行线程A，A释放全局锁后，A,B，C同时参与竞争全局锁，在多核情况下，由其他的线程需要等待信号，唤醒后才能竞争，会出现线程A经常抢占全局锁的问题。

而使用多进程虽然能够利用到多核，但是进程启动时间长，耗费内存多，数据共享也比线程麻烦得多。

这是python的历史遗留问题，因为引入GIL，许多用c语言写的扩展库不必考虑线程安全问题，很容易集成进来，使得python易于使用和扩展。

以上说的是基于cpython解释器的python，在jython中不存在GIL的设定。

**什么是协程**

又称微线程、纤程，优点：

1. 在单线程里实现任务的切换。是一种伪多线程，其不需要像线程一样上下文切换(协程在用户态执行)，开销远小于线程的开销。
2. 利用同步的方式去实现异步
3. 不再需要锁，提高了并发性能

p.s.进程由系统分配资源、线程是由CPU调度、协程由用户(程序)控制。

协程适合处理I/O密集型程序，当程序在执行I/O时操作时，CPU是空闲的，此时可以充分利用CPU的时间片来处理其他任务。在单线程中，一个函数调用，一般是从函数的第一行代码开始执行，结束于return语句、异常或者函数执行（也可以认为是隐式地返回了None）。有了协程，我们在函数的执行过程中，如果遇到了耗时的I/O操作，函数可以临时让出控制权(产生软件中断)，让CPU执行其他函数，等I/O操作执行完毕以后再收回控制权。

协程的发展：1）最初的生成器变形yield/send2）引入@asyncio.coroutine和yieldfrom3）在最近的Python3.5版本中引入async/await关键字。

t=yildindex通过send()返送的信息将赋值给t

**list、str、dict、set实现细节和区别**

**list**

是一个指针数组PyObject\*\*obitem。可以这么理解，链表的每一个节点是一个元素，而在list中，链表的每一个节点是一个数组。而且长度可变，所以一开始申请list的时候，会多申请一些冗余空间，减少重新分配的次数

**tuple**

也是一个指针数组，但是空间大小固定staticpyobject\*free\_list[Tuple\_SIZE].而且python还做了优化，e.g.小于一定大小的元组，在释放的时候会被放进这个free\_list中以供下次使用。也就是说，如果以后需要再去创建同样的tuple，Python就可以直接从缓存中载入。

**dict:**

字典底层维护一张哈希表，把哈希表看成一个列表，列表中的值存储了哈希值、键、值3个元素。E.g.entities=[[‘—‘,’—‘,’—'],[hash,key,value]…]

当插入一个键值对时，首先计算index=hash(key)&mask(mask为字典长度-1),若index已被占用，则判断两者key是否相等，相等就直接更新value值，不相等就继续向下寻找空位置(开放寻址法)。

p.s.entities是稀疏的，无序的，会动态扩展长度，每次的扩展都会重新计算所有key的hash值。

Python3.7以后，使用了indice表辅助字典，使得遍历字典时，按照插入的顺序遍历。

**set**

可变、无序。与字典的实现颇为相似，集合被实现为带有空值的字典，键是集合元素。

**frozenset**

不可变，无序

**为什么list不能当dict的key**

dict必须要保证当同样的内容进来的时候，能够找到该内容对应的值，也就是如果内容相同对应的哈希值应当相同。

而对于list的hash函数，可能存在两种实现方式：

1）基于id。基于id实现hash可以保证，两个对象的hash值不同，他们的id不同，但是,当两个对象的内容相同时，他们的id也可能不同，创建以个一模一样的list用字典查找永远会得到一个KeyError。

2)基于内容。tuple就是这样做的，tuple不可修改，基于内容方法对于同一个tuple来说其hash值不会改变，而对于同一个list而言，list的内容改变了，其hash值就改变了，因此，对于某一个list来说，基于内容的方法不可能生成一个固定的哈希值。

**tuple是不是冗余设计**

1）tuple不可变，可哈希，可作为字典的key

2）当给一个函数传值时，不想让该函数改变参数的值，使用tuple

3）tuple不可变，不需要申请冗余空间，而且python对tuple做了优化，会比list快一点点点。

**可迭代对象collections.abc.Iterable、迭代器Iterator、生成器Generator**

**哪些是可迭代对象**

字符串、列表、元组、集合、字典、文件

类中定义了\_\_iter\_\_的对象。

**为什么要有迭代器：**

字符串、列表、元组支持索引，我们可以使用while循环取出其中的元素，但是集合、字典、文件就不可以了，因此引入迭代器。

优点：1)为有序的和无序的类型提供了一种统一的迭代取值方式。

2)惰性计算，每迭代一次才会计算一次。这样使得迭代器可以表示一个计算流，对于迭代器来说，同一时刻，内存中只有一个值，所以可以给出无限量的数据流。而可迭代对象则需要将所有数据全部放入内存中。

缺点：1)除非迭代到抛StopIteration异常，否则无法知道迭代器长度。

2)同一时间只能取一个值，而且只能往前不能后退，若想再次取上次取到的值，就只能从头开始next。

iter(可迭代对象x)可以返回x对应的迭代器e.g.x为列表，就返回一个列表迭代器；若x不是可迭代对象，但是x内部实现了\_\_getitem\_\_，也可以，因为iter如果找不到\_\_iter\_\_,回去找\_\_getitem\_\_拿到第一个元素。

**关于for循环**

for循环又叫可迭代循环，首先for循环拿到一个可迭代对象，通过iter()将其转化成迭代器，迭代器通过next()拿到返回值，并将返回值赋值给对象，直到跑出StopIteration，for循环会捕获异常，结束循环。

**生成器既是可迭代对象也是迭代器**

1）列表生成器L=（xforxinrange(10)）

2）使用yield定义生成器函数:e.g.写一个斐波那契数列生成函数fib

python中虚拟机类似程序在x86机器上运行时栈的形式，以栈帧为基本单位，形成一个栈帧链，执行的时候在这些栈帧链中进行切换。在python中，一个模块、类以及函数的执行都会产生一个栈帧，然后执行这个栈帧。栈帧中保存了代码的信息和上下文，包括最后执行的指令、变量、异常状态等。

每次运行到yield的时候函数就主动挂起，交出运行权，并保存该函数的栈帧，下次再迭代的时候(使用next或send)，恢复栈帧中的数据。

p.s.迭代器、生成器中的值通过next()只能读一次

**内存溢出和内存泄漏的区别**

内存溢出outofmemory，是指程序在申请内存时，没有足够的内存空间供其使用，出现outofmemory；比如申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。

内存泄露memoryleak，是指程序在申请内存后，无法释放已申请的内存空间，并不是内存的消失，而是程序对某一块内存失去控制。一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

**直接赋值、深拷贝和浅拷贝**

赋值，传对象的引用（别名）

浅拷贝:拷贝父对象，不会拷贝对象的内部的子对象

深拷贝:copy模块的deepcopy方法，完全拷贝了父对象及其子对象

**可变类型和不可变类型**

可变：list、set、dict

不可变：int/float、str、tuple

**解释型语言和编译型语言的区别**

我们编程都是用的高级语言(写汇编和机器语言的大牛们除外)，计算机不能直接理解高级语言，只能理解和运行机器语言，所以必须要把高级语言翻译成机器语言，计算机才能运行。

编译型语言，在运行前进行翻译(编译)，这样运行时就可以直接执行了；

而解释型语言在运行的时候才翻译。

二者各有利弊：前者由于程序执行速度快，同等条件下对系统要求较低，因此像开发操作系统、大型应用程序、数据库系统等时都采用它，像C/C++、Pascal/ObjectPascal（Delphi）等都是编译语言，而一些网页脚本、服务器脚本及辅助开发接口这样的对速度要求不高、对不同系统平台间的兼容性有一定要求的程序则通常使用解释性语言，如JavaScript、VBScript、Perl、Python、Ruby、MATLAB等等。

编译型和解释型语言越来越笼统，主要体现在一些新兴的高级语言上，而解释型语言的自身特点也使得编译器厂商愿意花费更多成本来优化解释器，解释型语言性能超过编译型语言也是必然的。

**为什么说python比较慢？**

1. python是动态语言，事先不知道变量的类型
2. 是解释型语言而不是编译型语言
3. 其对象模型会导致访问内存的效率低下。比如说对于整数进行操作，python会有一个额外的类型信息层。

python优点:

1. python是动态语言，用法更加灵活和兼容。
2. python开源，跨平台，移植性强。
3. 很多需要用到c或fortran进行优化的场合，python都有强大的API或库进行支持。

**‘\_\_new\_\_’与‘\_\_init\_\_’的区别**

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

是在新式类中新出现的方法，是类的静态方法，调用\_\_new\_\_时没实力还不存在，\_\_new\_\_会返回一个实例，充当构造器的作用。

\_\_init\_\_(self,\*args,\*\*kwargs):

\_\_init\_\_初始化类的属性，不能有返回值。而在\_\_init\_\_启动之前，\_\_new\_\_决定是否要使用该\_\_init\_\_方法，因为\_\_new\_\_可以调用其他类的构造方法或者直接返回别的对象来作为本类的实例。

在旧类中没有\_\_new\_\_，\_\_init\_\_其构造器的作用。

**@classmethod和@staticmethod的区别，以及分别运用在哪些使用场景？**

1）classmethod的第一个参数为类本身(cls)，正如实例方法的第一个参数为对象本身(self);staticmethod第一个参数不需要传入cls或self，故staticmethod中是无法访问类和对象的数据的。

2）都可用类名直接调用；也可用实例对象调用（不推荐，没必要）

两者特点：

classmethod可以设置修改类属性；也可以实例化对象；

staticmethod无法访问类或对象的数据，所以可把它当作一个辅助功能方法用，里面包含一些与该类有关的逻辑代码。比如validate(\*args)🡪来对要初始化数据进行有效性检查。

**切片**

反转s[::-1]倒着，每隔两个取s[::-2]

**python2和python3的区别**

1. 编码py2默认asscii编码(8位2进制)，需要再顶部写#coding=utf-8(py2诞生的时代还没有unicode)，py3默认utf-8。
   1. Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。UTF-8是隶属于Unicode的可变长的编码方式。
   2. Unicode方式编码的字符串，可以使用encode()方法来编码成指定的bytes，也可以通过decode()方法来把bytes编码成字符串。
2. 输入python2input()只接受数字，raw\_input()接受字符串，python3中只有input()，接收字符串。
3. printpy2是把括号里的内容当成一个整体，而py3则是把字符串看成参数。
4. 字符串py2字符串有两种类型，unicode和str,分别表示文本字符串和字节序列，两者没有明显的界限，py3做了严格的区分，str表示字符串，byte表示字节序列，任何需要写入文本或网络的数据只接收字节序列，从源头阻止了编码错误的问题。

字节🡪存储、传输用字符🡪展示用

1. py2中True和Flase是两个全局变量，可被重新赋值；py3将其变为两个关键字，不可被赋值。
2. 整除py2和py3都认为//是整除,不同的是/,3/2python2是1,py3是1.5。3./2py2才是1.5。
3. 迭代器，py2中很多返回列表对象的内置函数和方法在py3都改成了但会类似于迭代器的对象，因为迭代器的惰性加载特性使得操作大数据更有效率。

e.g.字典dict.keys()dict.values()dict.items()pyt2返回列表，py3返回view对象，类似迭代器。mapfilter从函数变成了类，其返回结果从列表变成了可迭代对象。

1. nonlocal：global可用于函数内部修改全局变量的值，而在嵌套函数中想要声明一个全局变量是没办法实现的，py3新增nonlocal关键字，实现了该功能。

**pythonv.s.java**

都是解释性语言，都是强类型语言，万物皆对象。

python是动态语言，java是静态语言,也正是由于这样，python比java要慢一些，

java中的容器，比如数组，存放的变量类型必须一致，但python中的list变量类型任意。

java相对于相同类型，会安排连续的内存存放，寻址速度快；而python是全内存寻找变量，比java慢。

python提供了各种库，java没有这么多开源库，一般是发布jar包，看不到原始代码。

python的好多程序是面向过程的，而java主要是面向对象设计方法。

**高阶函数**

map(func,\*iterables)🡪map(lambdax:x\*x,[1,2,3,4])

filter(func,\*iterables)🡪map(lambdax:x>1,[1,2,3,4])

fromfunctoolsimportreduce

reduce(function,sequence,initial=None)🡪reduce(lambdax,y:x+y,[1,2,3,4,5])🡪((((1+2)+3)+4)+5)=15reduce(lambdax,y:x+y,[1,2,3,4,5],10)🡪25

**is与==的区别**

is比id,==比值。

**pass**

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，使程序不报错，不会进行任何操作。

**python中的参数类型**

位置参数、默认参数、可变参数（\*arg）、关键字参数(\*\*kargs)。

\*arg,\*\*kargs都是为了实现任意个参数的方式。\*arg会把位置参数转化为tuple，\*\*kargs会把关键字参数转化为dict。

p.s.**默认参数必须指向不变类型**e.g.None，如果默认参数指向[]，会造成严重bug!!!

**单下划线和双下划线的区别**

\_\_foo\_\_:定义的是特殊方法，一般是系统定义名字，防止与用户定义的变量名冲突，类似\_\_init\_\_()之类的。

\_foo:以单下划线开头的表示的是protected类型的变量，即保护类型只能允许其本身与子类进行访问，不能用于frommoduleimport\*

\_\_foo:双下划线的表示的是私有类型(private)的变量,只能是允许这个类本身进行访问了。如果子类想要访问父类中的私有变量或方法，可以使用实例.\_父类名\_\_foo

foo:就是public方法

**python弱引用**

在不增加对象的引用计数个数的情况下获得对象的引用

**重写和重载的区别**

重写是子类对父类的允许访问的方法的实现过程进行重新编写,返回值和形参都不能改变。即外壳不变，核心重写！

重载(overloading)是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

**上下文管理器**

e.g.withopen(‘…’,’r’)asf:withthread\_lock:

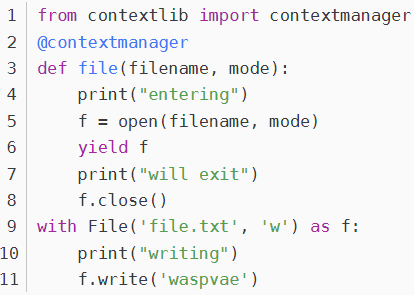
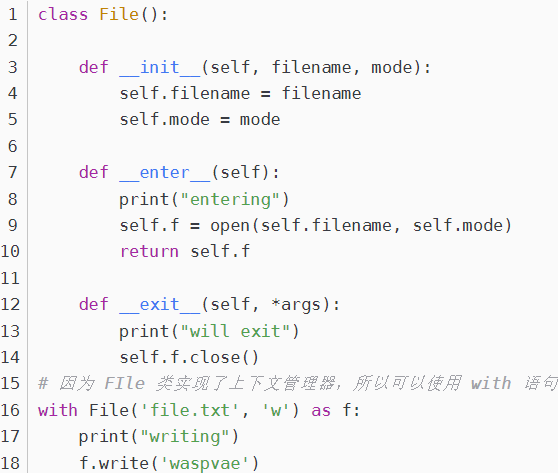
线程池也是:\_\_enter\_\_(self):returnself\_\_exit\_\_(self,exc\_type,exc\_val,exc\_tb):self.terminate()

任何实现了\_\_enter\_\_和\_\_exit\_\_方法的对象都可被称为上下文管理器，上下文管理器对象可以使用with关键字。

1）with…as，紧跟with后面的语句被求值后，返回对象的\_\_enter\_\_方法，并将该方法的返回结果绑定到as后面的变量；当with后面的代码块全部被执行完以后，调用前面返回对象的\_\_exit\_\_方法。exit方法被执行。开发库时，清理资源、关闭文件等操作，都可放在exit方法中。简化了每次写finally的工作。实际上，with后面代码块抛异常时，上下文管理器会捕获异常，exit方法的参数中包括异常类型、异常值、异常堆栈信息，若没有报异常，则这三个参数将都为None。

2）把生成器函数当成上下文管理器来使用，

contextlib.contextmanager装饰器@contextmanager



两个代码等价，通过yield将函数分割成两部分，yield之前的语句在enter方法中执行，yield之后的语句在exit方法中执行。紧跟在yield后面的值是函数的返回值。

**强类型、弱类型、动态语言、静态语言**

**动态语言：**运行期间才做数据类型检查的语言，即动态类型语言编程时，永远不用给任何变量指定数据类型。该语言会在第一次赋值给变量时，在内部将数据类型记录下来。E.g.pythonjavascript。

优点：方便阅读，不需要写非常多的类型相关的代码；

缺点：不方便调试，命名不规范时会造成读不懂，不利于理解等

**静态语言**：编译期间做检查数据类型的语言，即写程序时要声明所有变量的数据类型，是固定的。使用数据之前，必须先声明数据类型（int,float,double等）。相当于使用之前，首先要为它们分配好内存空间。E.g.cc++c#java

**强类型定义语言**：一旦变量被指定某个数据类型，如果不经强制转换，即永远是此数据类型。

**弱类型定义语言：**一个变量可以赋不同数据类型的值。

Python属于强类型、动态语言。

例如，python不能合并类型‘x’+3。而在javascript中‘x’+3会变成‘x3’。

因为强类型化意味着值的类型不会以意外的方式改变。只包含数字的字符串不会神奇的变成数字，每种类型的更改需要显示的转换。Python能够有如下操作：

a=1

a=‘test’

是因为python是动态语言，动态类型化意味着运行到赋值语言时对象才具有类型。这与C语言的旧版本形成对比，c是弱静态类型，指针和证书几乎可以互换。

静态类型语言在如下操作下会出错：

inti=1;

i=“1”;#会报错

**鸭子类型**

鸭子类型是编程语言中动态类型语言中的一种设计风格，一个对象的特征不是由父类决定(静态语言中是这样)，而是通过对象的方法决定的。e.g.任何实现了\_\_iter\_\_和\_\_next\_\_方法的对象都可称之为迭代器，但对象本身是什么类型不受限制，可以自定义为任何类。

我们并不需要继承Iterator就可以实现迭代器的功能。E.g.有一函数希望接收的参数是Iterator类型时，但是我们传递的是Foo的实例对象，其实也没问题，换成是Java等静态语言，就必须传递Iterator或者是它的子类。

鸭子类型通常得益于"不"测试方法和函数中参数的类型，而是依赖文档、清晰的代码和测试来确保正确使用，这既是优点也是缺点，缺点是需要通过文档才能知道参数类型，为了弥补这方面的不足，Python3.6引入了类型信息，定义变量的时候可以指定类型:

defgreeting(name:str)->str:

return'Hello'+name

**元类**

当我们使用关键字class时，python解释器在执行时就会自动的创建一个对象。

我们也可以手动处理，使用内建函数type。type两种使用方法：

1）type(‘1’)🡪返回参数的类的描述

2）type(类名，父类的元组(可以为空)，包含属性的字典名称:值)

classFoo(object)

pass

Foo=type(‘Foo’,(),{})🡪返回一个**类对象**

Foo=Foo()🡪返回一个**类的实例**

type就是一个元类，就是python在背后用来创建所有类的元类：

a=1

a.\_\_class\_\_🡪<type‘int’>

a.\_\_class\_\_.\_\_class\_\_🡪<type‘type’>

**如何自己创建元类呢？**

在类中定义\_\_metaclass\_\_=…

python做了如下操作：

Foo中有\_\_metaclass\_\_这个属性吗？如果是，Python会在内存中通过\_\_metaclass\_\_创建一个名字为Foo的类对象（我说的是类对象，请紧跟我的思路）。如果Python没有找到\_\_metaclass\_\_，它会继续在父类中寻找\_\_metaclass\_\_属性，并尝试做和前面同样的操作。如果Python在任何父类中都找不到\_\_metaclass\_\_，它就会在模块层次中去寻找\_\_metaclass\_\_，并尝试做同样的操作。如果还是找不到\_\_metaclass\_\_,Python就会用内置的type来创建这个类对象。

**python多继承的顺序是什么**

python3都是新式类，python2中没有显示继承object，就不是新式类，旧类并不是默认继承object类，而是继承type类。。

classD(object)classC1(D)classC2(D)classB(C1,C2)

classA(B,C1,C2)的继承顺序是什么？

新式类：广度优先搜索，A🡪B🡪C1🡪C2🡪D🡪object

经典类：从左到右，深度优先搜索，A🡪B🡪C1🡪D🡪object🡪C2

**python中的一些文件**

.pyc文件是python的字节码文件。

.pyc文件是不能用文本编辑器进行编辑的。.pyc文件的内容与平台无关。其优点是.pyc文件的执行速度要远快于.py文件。至于为什么要有.pyc文件，这个需求太明显了，因为.py文件是可直接看到源码的，若是软件开发商的话，不可能把源码泄漏出去？所以，就需编译成.pyc后再发布。.pyc文件默认情况下不会自动生成。若你在命令行直接输入“pythonpath/to/projectDir”（假设projectDir目录含有“\_\_main\_\_.py”文件，以及其他将要调用的模块），那么程序运行结束后便自动为当前目录下所有的脚本生成字节码文件，并保存于本地新文件夹\_\_pycache\_\_当中。

PyCodeObject与.pyc文件 PyCodeObject是Python编译器真正编译成的结果。1)当python程序运行时，编译的结果保存在位于内存中的PyCodeObject中，当Python程序运行结束时，Python解释器则将PyCodeObject写回到pyc文件中，前提有写权限。2)当python程序第二次运行时，首先程序会在硬盘中寻找pyc文件，如果找到对比修改时间，改动了就重新编译，没改动则直接载入，否则就重复上面的过程。所以我们应该这样来定位PyCodeObject和pyc文件，我们说pyc文件其实是PyCodeObject的一种持久化保存方式。

.pyd文件并不是用python编写成的，.pyd文件一般是其他语言编写的python扩展模块。.pyd文件是用D语言按照一定格式编写，并处理成二进制的文件。

**多线程的实现方式**

def func(x):return x+1

直接使用t=threading.Thread(target=func,args=(1,))

或者定义一个新的类MyThread继承Thread，且必须重写run方法。

一些常用方法：

**t.setDeamon(True)或t.deamon=True**把t线程设置为守护线程，必须在t.start()之前设置。守护线程是一种后台线程，当主线程结束，守护线程也会结束。e.g.python垃圾回收线程。

**t.setName(s)或t.name=s**设置线程名

**t.start()**开启一个新的线程，线程名就是我们定义的name，该线程进入到等待队列，等待cpu时间，当其获取到cpu时间，会有一个专门的控制线程调用run(),一个线程只能执行一次start()。

**run()**并不启动一个新线程，就是在主线程中调用了一个普通函数而已。

**time.sleep(seconds)**

**t.join(timeout=None)**A线程调用了t线程的join()，A线程会阻塞，直到t线程执行结束。或A线程阻塞timeout秒。

**t.is\_alive()**判断线程是都存活。线程的存活时间为start()--》终止

**多线程的同步**

互斥锁：保证多线程共享变量的问题

**Lock**lock=threading.Lock()

lock.acquire()获取锁。未获取到会阻塞程序，称道获取到锁才会往下执行。

lock.release()释放锁。acquire和release必须成对出现，否则可能会造成死锁。

为避免着用情况，使用withlock:..更为优雅

**RLock(可重入锁)**threading.RLock()在同一线程中，放松对锁钥匙的获取，其他同Lock一致。RLock可以锁两次，重进入。

死锁发生，一般是同一线程嵌套获取同把锁或多个线程不按顺序同时获取多把锁，前者重入锁可以避免。对于后者，数学证明，只要两个（或多个）线程获取嵌套锁时，按照固定顺序就能保证程序不会进入死锁状态。（可以重写acquire,在其中将所有锁按顺序排序，总是按照顺序获取所有锁，并重写release倒序释放锁）。

**Semaphore**Lock的加强版，可以被多个线程同时拥有，而Lock只能被某一个线程拥有。semap=threading.Semaphore(value=1)semap.acquire()如果多个acquire()调用被阻塞，release()将恰好唤醒一个，可以随机，可以顺序。

**多线程信息通信机制**

**Event:**多个线程可以等待某事件的发生，当事件发生后，所有的线程都会被激活。

threading.Event()

event.clear()重置event，使得所有该event事件都处于待命状态。

event.wait()等待接收event的指令，决定是否阻塞程序执行。即等到event.set()后才会继续执行。

event.set()发送event指令，使所有设置该event事件的线程执行。

event.is\_set()

**Condition**,类似于Event

threading.Condition()

cond.acquire()类似lock.acquire()

cond.release()类似lock.release()

cond.wait(timeout=None)等待指令触发，同时会释放对锁的获取，直到被notify才重新占有锁。

cond.notify(n=1)/cond.notify\_all()发送指令，触发执行。

**Queue**从一个线程向另一个线程发送数据最安全的方式可能就是使用queue库中的队列了。创建一个被多个线程共享的Queue对象，这些线程通过使用put()和get()操作来向队列中添加或者删除元素。

fromqueueimportQueue(先进先出)/LifoQueue(后进后出)/PriorityQueue(优先级队列)

Queue(maxsize=0)默认大小不受限制

q.get(timeout=None)阻塞程序，等待队列消息

q.put(s)发送消息q.put(消息,优先级)

q.join()等待所有的消息被消费完

q.qsize()查询当前队列的消息个数

q.empty()

q.full()

e.g.老师点名

**想要在当前线程中创建一个全局变量，并且各自线程互不干扰？**

fromthreadingimportlocal

local\_data=local()

local\_data.自定义字段=…

**线程池**

在使用多线程处理任务时也不是线程越多越好，由于在切换线程的时候，需要切换上下文环境，依然会造成cpu的大量开销。

1. 系统启动一个新线程的成本是比较高的，因为它涉及与操作系统的交互。在这种情形下，使用线程池可以很好地提升性能，尤其是当程序中需要创建大量生存期很短暂的线程时，更应该考虑使用线程池。
2. 线程池在系统启动时即创建大量空闲的线程，程序只要将一个函数提交给线程池，线程池就会启动一个空闲的线程来执行它。当该函数执行结束后，该线程并不会死亡，而是再次返回到线程池中变成空闲状态，等待执行下一个函数。
3. 此外，使用线程池可以有效地控制系统中并发线程的数量。当系统中包含有大量的并发线程时，会导致系统性能急剧下降，甚至导致Python解释器崩溃，而线程池的最大线程数参数可以控制系统中并发线程的数量不超过此数。
4. 如果一个线程的时间非常长，就没必要用线程池了。
5. 我们不能控制线程池中线程的开始、挂起、和中止。

**p.s.**线程池可以看做容纳线程的容器；一个应用程序最多只能有一个线程池。

ThreadPool更适合并发运行若干个运行时间不长且互不干扰的函数。

**pool=multiprocessing.pool.ThreadPool(thread\_num)**

**result=pool.apply(func=func,args=(...),callback=...,error\_callback)**同步方式执行，如果指定了回调函数，则它应该是可调用的，它接受单个参数。当结果变为就绪时，将对其应用回调，即除非调用失败，在这种情况下将应用error\_callback。如果指定了error\_callback，那么它应该是一个可调用的，它接受一个参数。如果目标函数失败，则使用异常实例调用error\_callback。回调应立即完成，否则处理结果的线程将被阻止。

**result=pool.apply\_async(func=func,args=(...),callback=...,error\_callback)**异步方式。

**result.get(timeout=None)**同步执行,所以不能直接在apply\_async后使用，否则会造成顺序执行，可以先把result加入到一个列表中，apply\_async所有进程后，再统一get。**pool.close()**调用join之前，先调用close或terminate函数，否则会出错。执行完close后不会有新的进程加入到pool。

**pool.terminate()**close()会等待池中的工作进程执行结束再关闭pool,而terminate()则是直接关闭，不再处理未处理的任务。

**pool.join()**等待所有子进程结束。

**应用场景：**

1）**高并发、任务执行时间短**的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换；

2）**并发不高、任务执行时间长**的业务要区分开看：

a）假如是业务时间长集中在IO操作上，也就是**IO密集型的任务**，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务；

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换，cpython不适合用多线程。

3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

**守护线程**

在创建新线程时，子线程会从其父线程继承其线程属性，主线程是普通的非守护线程，默认情况下，它所创建的任何线程都是非守护线程。在这种情况下，如果新线程在运行，主线程将永远等待，无法正常退出。

我们经常需要创建线程来执行某项例行任务或提供某种特殊服务，常见的例子就是垃圾收集器。我们希望这些线程在主线程结束时，也会结束，就可以将他们设置为守护进程：

1.必须在启动之前将线程配置为守护程序或非守护程序，否则Python将引发运行时错误；

2.最后守护程序线程不会像普通线程一样正常退出，当程序中的所有非守护程序线程都完成执行时，任何剩余的守护程序线程将在Python退出时被放弃，在设计守护线程时，需要确保在主线程退出时不会产生任何负面影响。

**运行一个py文件,分为三个步骤**

1.从硬盘加载Python解释器到内存

2.从硬盘加载py文件到内存

3.解释器解析py文件内容，交给CPU执行

其次需要明确的是每当执行一个py文件，就会立即启动一个python解释器

开启子线程时，给子线程指定了一个target表示该子线程要处理的任务即要执行的代码。代码要执行则必须交由解释器，即多个线程之间就需要共享解释器，为了避免共享带来的数据竞争问题，于是就给解释器加上了互斥锁！

GIL的加锁与解锁时机

加锁的时机：在调用解释器时立即加锁

解锁时机：-当前线程遇到了IO时释放

-当前线程执行时间超过设定值时释放

GIL保护的是解释器级别的数据安全，比如对象的引用计数，垃圾分代数据等等，具体参考垃圾回收机制详解。对于程序中自己定义的数据则没有任何的保护效果，这一点在没有介绍GIL前我们就已经知道了，所以当程序中出现了共享自定义的数据时就要自己加锁。

**同步异步-阻塞非阻塞**

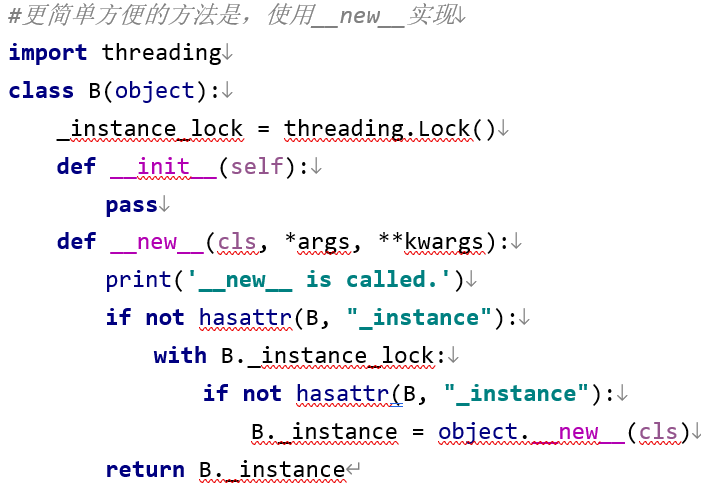
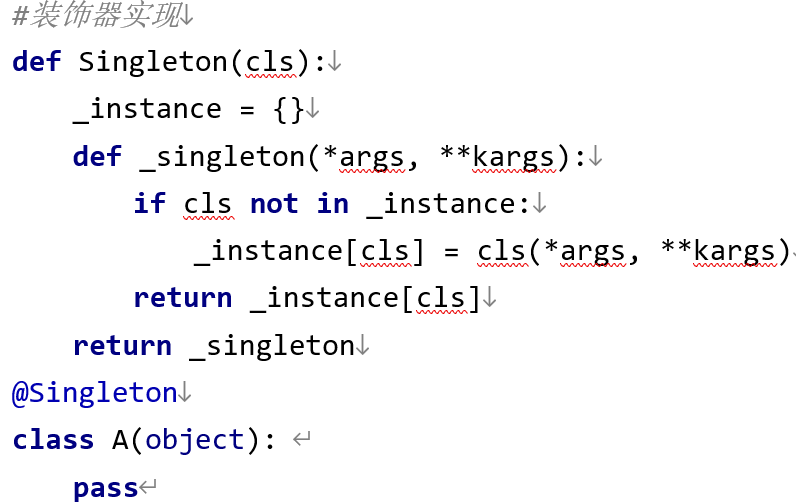
阻塞：当程序执行过程中遇到了IO操作，在执行IO操作时，程序无法继续执行其他代码。非阻塞：程序在正常运行没有遇到IO操作，或者通过某种方式使程序即时遇到了也不会停在原地，还可以执行其他操作，以提高CPU的占用率。

同步指调用：发起任务后必须在原地等待任务执行完成，才能继续执行。

异步指调用：发起任务后必须不用等待任务执行，可以立即开启执行其他操作。

同步会有等待的效果但是这和阻塞是完全不同的，阻塞时程序会被剥夺CPU执行权，而同步调用则不会！很明显异步调用效率更高。

**单例模式**



懒汉式（需要对象的时候才加载）

饿汉式(预先创建对象)

**优点：**只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等

**缺点：**不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。

滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

**应用实例**

打印机

很多软件会有一个或多个属性文件存储系统配置，应有一个对象去管理这些书信该文件。

windows的任务管理器、回收站。 网站的计数器 应用程序的日志。 数据库连接池，属于占用资源较多的对象。 线程池、进程池。 操作系统的文件系统，最大的单例模式的例子。

**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。

**工厂模式**

**定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。**

一个Shape类，定义一个抽象函数draw(self) @abc.abstractmethod，有n个子类包括圆形、三角形……

一个ShapeFactory类，实现一个静态方法get\_shape(shape\_type) @staticmethod 根据shape\_type的值，创建不同的shape子类。

在使用时，若想要创建三角形，只需ShapeFactory.get\_shape(‘Triangle’).draw()即可.

**优点：**1.无需知道创建的对象需要的类名；2.可以传入参数，指定创建的对象。

**缺点**：每增加一个形状，都要增加一个具体类并修改ShapeFactory。

**抽象工厂模式**

**提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。**

一个AbstractFactory:包含两个抽象函数get\_shape和get\_color

ShapeFactory继承AbstractFactory,重写get\_shape(shape\_type)，根据不同的shape\_type创建不同的shape;在重写get\_color(color\_type)时，返回None。ShapeFactory下有n个子类，包括Cicle、Triangle……

ColorFactory……

定义一个FactoryProducer,实现一个静态函数getFactory(factory\_type),根据不同的factory\_type值，生成ShapeFactory或ColorFactory.

在使用时，FactoryProducer.getFactory(‘Shape’).get\_shape(‘Cicle’).draw()

**优点：**当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

**缺点：**产品族扩展非常困难，要增加一个系列的某一产品，既要在AbStractFactory里加代码，又要在具体的里面加代码。

**桥接模式**

假设有形状（圆形、三角形等m中形状）、颜色（红黄蓝等n中颜色）

桥接模式前：要创建红色圆形类、黄色圆形类等m\*n个类。这是由于形状和颜色两个属性的高度耦合，造成类的数量特别多，而且不容易扩展。

桥接模式用来实现二者的解耦。

定义一个shape类，init参数为 行传(‘cicle,’triangle’……)shape下有m个子类，

并定义一个color类, init参数为 颜色(‘red’,’yellow’……)，color下有n个子类。

可以在shape类下定义一个颜色变量c，draw函数用来绘制x颜色的y形状，就可以调用self.c.颜色和self.形状。

需要创建(m+1)+(n+1)个类，且更容易扩展。

**优点：**1、抽象和实现的分离。 2、优秀的扩展能力。 3、实现细节对客户透明。

**缺点：**桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

**观察者模式**

对象之间存在一对多的依赖关系时，可使用观察者模式。比如，当一个对象被修改时，需要自动通知依赖它的对象。