**第一章：基本语法**

Range和xrange的区别

两者用法相同，不同的是 range 返回的结果是一个列表，而 xrange 的结果是一个生成器，前者是直接开辟一块内存空间来保存列表，后者是边循环边使用，只有使用时才会开辟内存空间，所以当列表很长时，使用 xrange 性能要比 range 好。

字典是可变对象，所以对字典进行操作时，如果是传参，放进去的是引用的话，那么相当于浅拷贝，实际上所有的值都会变成最后一个数的。

4g内存怎么读取一个5g的数据：

方法一：可以通过生成器，分多次读取每次读取数量相对较少的数据进行处理，依次读取。

方法二：可以通过linux命令split切割成小文件，然后再对数据进行处理，此方法效率比较高。可以按行数

内存只有4G无法一次性读取10g的文件，分批读入，分批读入数据要记录每次读入数据的位置。分批每次读入数据的大小，太小就会在读取操作上花费过多时间。

Readlines（）

返回 yield

Read 读取整个文件

Readline 读取下一行，使用生成器方法

Readlines 读取整个文件到一个迭代器以供我们遍历

Except中return后会执行finally中的代码，

Raise可以抛出自定义异常

python有自动回收垃圾的机制

多个引用指向一个对象的时候，有一个计数器维护着这个对象的内存地址，删除一个应用，则删除一个指向对象的连接，计数器减1，直到计数器减到0的时候。则python的垃圾回收机制会自动的将内存地址回收，此时会调用\_\_del\_\_()

对象赋值就是简单的对象引用，赋值操作不会开辟新的内存空间，他只是复制了对象的引用。也就是说了出了b这个名字以外，没有其他的内存开销。

浅拷贝拷贝了指向原对象的引用，也会创建新对象，其实修改b是不会影响到列表a的，

a[3].append(‘java’) 就影响了，append操作是能影响到浅拷贝的，拷贝的对象也会发生变化的，修改了嵌套的list，修改外层元素，会修改它的应用。

深拷贝则是拷贝了对象的所有元素，包括了多层嵌套的元素。两者不会产生影响，因为拷贝出来的是一个全新的对象。与原来的对象没有关联。

导入copy模块，deepcopy（）是深拷贝，所以id（c）和id（e）的地址不一样，copy（）则是视拷贝数据类型不同而变，如果拷贝的是不可变数据类型，比如元组，那么拷贝就是内存地址的链接，很好理解，因为不可变数据类型本身就是不能更改的，拷贝内存地址也没有意义，但是拷贝的是可变数据类型，比如列表的时候，那么就是拷贝内存地址，但是嵌套层只拷贝链接，这是与深拷贝不同的地方，深拷贝从嵌套出来全都都是拷贝内存地址。

不可变数据类型深浅拷贝一样的，但是可变数据类型就不一样了。

\_\_init\_\_在创建对象后，对对象进行初始化

new 是在对象创建之前创建一个对象，并将该对象返回给init

python是强类型的动态脚本语言

不允许不同类型相加

不使用显示数据类型声明，且确定一个变量的类型是在第一次给它赋值的时候。

解释性语言，不需要编译

可以使用目标类型关键字强制类型转换

可以使用内置函数直接转换的有 tuple(list)

Python中的string有str和unicode两种，python3只支持unicode的string。

所有有一个encode和decode的编码格式转换

Python2中，For循环会修改外部相同名称变量的值。Python3中，for循环不会更改外部相同名称变量的值。

提升python运行的性能

多使用多进程，利用多核性能；用C重写部分代码；IO多路复用；使用python的内建函数；

使用局部变量

**第二章 数据类型**

字典是键值对，无序的

可变不可变值得是内存中的值是否可以被改变，不可变类型值指的是内存块里面的值不可以改变，有数值、字符串、元组；可变类型则是可以改变，主要有列表、字典。

列表

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

带有默认参数表达式在函数定义时被计算，不在函数调用时计算。不往默认值里面传参数的话，那么默认的会一直使用，但是传了参数就不再使用默认值了。

元组 不能修改，通过索引进行查找

集合 不重复

**第三章 Python高级**

上下文管理器

Python中提供了上下文管理器（Context Manager）的概念，可以通过上下文管理器来定义/控制代码块执行前的准备动作，以及执行后的收尾动作。

本文介绍了Python中的上下文管理器，以及如何结合with语句来使用上下文管理器。

总结一下with 语句的执行流程：

执行context\_expr 以获取上下文管理器对象

调用上下文管理器的 \_\_enter\_\_() 方法

如果有 as var 从句，则将 \_\_enter\_\_() 方法的返回值赋给 var

执行代码块 with\_suite

调用上下文管理器的 \_\_exit\_\_() 方法，如果 with\_suite 产生异常，那么该异常的 type、value 和 traceback 会作为参数传给 \_\_exit\_\_()，否则传三个 None

如果 with\_suite 产生异常，并且 \_\_exit\_\_() 的返回值等于 False，那么这个异常将被重新抛出到上层

如果 with\_suite 产生异常，兵器 \_\_exit\_\_() 的返回值等于 True，那么这个异常就被忽略，继续执行后面的代码

在很多情况下，with语句可以简化代码，并增加代码的健壮性。

Python中的类方法、类实例方法、静态方法

@classmethod

@staticmethod

类方法类和实例都是可以调用的，但是类实例方法只有实例化对象可以调用。静态方法和类没有太大关系，对象可直接调用

动态获取和设置对象的属性

Hasattr，getattr，setattr

内存管理与垃圾回收机制

**引用计数、垃圾回收、内存池**

Python的内存管理机制及调优手段？

引用计数：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计数时其计数减1，当引用计数等于0时删除。

垃圾回收

**引用计数**：python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。出现循环引用的话，引用计数机制就不起作用了。

**标记清除**：如果两个对象的引用计数都为1，但是仅仅存在他们之间的循环引用，那么这两个对象都是需要被回收的，也就是说，他们的引用计数虽然表现为非0，但是有效的引用计数为0。所以先将循环引用摘掉，就会得出这两个对象的有效计数。

**分代回收:** 从前面“标记-清除”这样的垃圾收集机制来看，与总的内存块的数量是相关的，当需要回收的内存块越多时，垃圾检测带来的额外操作就越多，垃圾回收带来的额外操作就越少。

**内存池**：

-1，-2 层主要是由操作系统进行操作

0层是malloc，free等内存分配和释放函数进行操作

1,2层是内存池，有python接口函数，当对象小于256k时由该层分配内存

3层是最上层，也就是我们直接对python对象的直接操作

Python在运行期间会大量的执行malloc，free的操作，频繁的在用户态和核心态之间进行切换，这将严重影响python的执行效率，为了加速python的执行效率，引用了内存池机制，用于管理对小块内存的申请和释放。大小内存的分界点定在256个字节，小于256k，会在内存池中申请内存。

经由内存池登记的内存到最后还是会回收到内存池,并不会调用 C 的 free 释放掉.以便下次使用.对于简单的Python对象

调优手段：

手动回收垃圾

调高垃圾回收阈值

避免循环引用

**内存泄露**是什么？

指由于疏忽或者错误造成程序未能释放已经不再使用的内存的情况。内存泄露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，失去了对该段内存的控制，因而产生了内存的浪费。

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

函数

不可变参数是拷贝值传递

可变参数是引用传递的

缺省函数

在调用函数的时候没有传入参数的情况下，在调用函数的同时赋值，赋值会代替默认参数。

递归函数的内部有一个条件判断

回调函数是要把函数的地址作为参数传递给另一个函数，将整个函数当做一个对象，赋值给调用的函数。

Lambda函数是一个可以接受任意多个参数并且返回单个表达式值的函数。

匿名函数简化代码

Python闭包的延迟绑定

设计模式

单例：某一个类只有一个实例存在

装饰器

装饰器本质是一个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的功能**，装饰器的返回值也是一个函数对象。

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继续使用。

闭包

在函数内部在定义函数，并且这个函数用了外面函数的变量，那么将这个函数以及用到一些变量称之为闭包。

迭代器有next方法和iter方法返回它自身，for循环调用了iter方法，iter返回一个定义了next方法的的迭代器对象，它在容器中逐个访问容器内的元素，next()也是python的内置函数。

生成器(Generator)，不想一次性生成所有的变量，因为有可能列表直接崩掉，生成器就是先生成链接，如果有的时候再调出来了。既想生成值，又不想占用太多的内存空间。

在需要返回数据的时候使用yield语句

Yield就是保存当前程序执行状态。

生成器用yield函数，用一次算一次，可以节省内存。生成器每次计算需要上一次计算结果，所以使用yield，否则一return，上次计算就没了。

**面向对象**

不可变对象，该对象所指向的内存的值不能改变。当改变某个变量的时候，由于其所指的值不能被该变量，相当于把原来的值在复制一份后再改变，这会开辟一个新的地址，变量再指向这个地址。

可变对象，引用改变之后，实际上所指的值直接发生改变，并没有发生复制行为，也没有开辟新的地址

数值、字符串、元组都是不可变类型。列表、字典和集合都是可变类型。

Is判断的是id，==判断的是对象的值是否相等

Python中的魔法方法

当对象实现了这些方法中某一个，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。

以两个下划线包围来命名的

\_\_init\_\_构造器，当一个实例被创建的时候初始化的方法

\_\_new\_\_

\_\_call\_\_ 允许一个类的实例就像函数一样被调用

对象私有化方法

Proprty把访问函数转换成访问属性了

**系统编程**

调度算法：子进程先还是主进程取决于调度算法，是相互交叉的

1. 时间片轮转
2. 优先级调度

并发：看上去是同时执行，实际上不是

并行：实际上就是同时执行，5个核执行5个程序

进程：程序运行在操作系统上一个实例，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：

1. 首先要导入multiprocessing中的process；
2. 创建一个Process对象
3. 创建Process对象时，可以传递参数
4. 使用start( )启动进程
5. 结束进程

进程池

主进程一般都是用来等待，真正的任务都是在子进程中执行的，但是进程数并不是越大越好

进程之间的通信 Queue

进程与进程之间是没有关系的，通信通过队列来实现

一个运行的程序就是一个进程，线程是调度执行的最小单位

多个进程就是多个资源，进程是资源分配的单位，线程是CPU调度的单位，进程已经分配了资源模块，线程执行具体运行

协程的调度是由用户控制的

锁是对线程控制的对象。

GIL全局解释锁

限制多线程同时执行，保证同一时间只有一个线程执行，python中的多线程是假的多线程，但是多进程不一样，多进程的效率要比多线程高的多。‘’

同步：多个任务之间有先后顺序的，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的，可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，这就叫回调。

阻塞：如果卡住了调用者，可以继续执行

非阻塞：

Python中进程与线程的使用场景

多进程适合在CPU密集型操作

多线程适合在IO密集型操作

线程是并发，进程是并行，进程之间是像话独立，是系统分配资源的最小单位，同一个进程中的所有线程共享资源。

网络编程