1. Python基础
2. break continue pass:

break跳出循环，continue进入下一层循环（有的话），pass跳过本行继续执行剩下的

1. 匿名函数

有些简单的函数方法不用显式的定义，直接用lambda关键字就行

1. %符号

运算符->取模 / 格式化输出

- 字符串输出 %s (+/- -> 右对齐/左对齐) .2(保留两位)

- 整数输出 %d 十进制 / %x 十六进制

- 浮点数输出 %f / %e 指数形式

- format() 功能更强大, 可以重复输出 / 可以指定顺序

1. \*args 星号将参数以元组形式导入

\*\*kwargs 双星号（\*\*）将多个参数以字典的形式导入

1. 为什么python比其他语言运行慢

解释性语言 / 动态语言 / 垃圾回收 / GIL / 一切都是对象

编译性语言和解释性语言:

编译：程序执行前把代码直接翻译成为机器码，确定数据类型，由CPU执行，快

解释：程序执行时解释器逐行执行代码，再确定数据类型，慢

1. 异常处理

try (可能出现异常的语句) except(异常发生后执行) else(无异常发生后执行)

finally(无论有无异常都执行) raise 主动触发异常

1. Python2/3 区别

- range() 函数 python2返回列表 python3返回可迭代对象(节省内存)

1. 常见的字符串方法

- s.find(str) # s中是否有子字符串str

1. Python编程的特点

- 解释性语言（脚本语言），直接运行代码，遇到错误报出

- 动态语言，运行时确定数据类型

- 面向对象语言，把一切视为对象，也支持函数式编程

- 简单、明了、高级、库丰富

1. Python和C的区别

语言类型 - C是编译性语言，生成机器码，CPU直接执行，速度快 / Python是解释性语言，逐行读取代码

Python 动态 / C 静态

内存管理 - Python垃圾自动回收机制 / C语言需要手动释放 malloc() free()

应用 - Python面向对象，开发高级软件 / C面向过程语言，底层架构开发，硬件开发

语法 - Python严格缩进，但是没有分号结尾等等

1. 面向过程和面向对象

面向过程 - 把问题拆分成一个个过程，用函数一步步实现 优点：复用性高，维护起来方便 缺点：低耦合

面向对象 - 把构成问题的事物拆分成一个个对象，用方法/属性描述对象在解决整个问题中的行为 优点：效率高，耦合程度高 缺点：复用性差，维护起来复杂

面向对象的三大特征

- 封装 把客观的事物抽象成为类和对象，实现一些属性和方法，并对不可信的其他对象隐藏

- 继承 可以使用父类的所有功能，可以进行拓展

- 多态 通过覆写方法，相同的实例方法对于不同类的对象有不同的实现

1. 魔法方法:

常见魔法方法:

双下划线(\_\_)开头和结尾

如\_\_init\_\_，\_\_call\_\_，\_\_iter\_\_，\_\_next\_\_

\_\_call\_\_ 类中定义，表示可调用对象

if \_\_name\_\_ == “\_\_main\_\_”的作用:

\_\_name\_\_是解释器自动创建的变量，默认\_\_main\_\_

如果有其他文件引用了当前文件里的函数，且我们在当前文件调用了相同的函数， 这个方法可以保证函数不会被重复执行

\_\_init\_\_和\_\_new\_\_的区别:

先调用new后调用init

\_\_init\_\_ 是实例方法,初始化类的对象时会调用，给对象赋值等等

\_\_new\_\_是类方法,return object.\_\_new\_\_(cls) 单例模式用到

1. Python面向对象编程
   1. 类变量和实例变量
      1. 类变量: 每个类只有一份
      2. 实例变量: 每个实例都有一份
   2. 类方法，实例方法和静态方法
      1. 实例方法: self参数 只能访问实例变量
      2. 类方法: @classmethod cls参数 可以访问类变量和实例变量
      3. 静态方法: @staticmethod 不可以访问类变量和实例变量 独立
2. 全局变量和局部变量
   1. 存储方式 全局变量存储在静态数据库，局部变量存储在堆栈上
3. Python进阶
4. 迭代器

迭代-访问集合元素的一种方式

iter() 返回迭代器，可迭代对象记录遍历顺序 next() 迭代方法

for循环本质就是迭代器+异常处理(Stop Iteration错误)

\_\_iter\_\_ 方法 标记类为可迭代对象/ 调用iter()方法时自动调用

\_\_next\_\_ 方法 标记当前类为迭代器/ 调用next()方法时自动调用

迭代器本身也是可迭代对象

1. 生成器

与列表比较:

只存储生成数据的算法->减少内存浪费 / 列表->存储所有数据,简洁明了

适合存储大量/不确定数量数据,但复杂 / 适合存储少量数据

与迭代器关系:

生成器能生成下一组数据，说明它一定存储上一组数据

这种迭代更新的思想表明 生成器就是一种特殊的迭代器，只是定义方式不同。

定义方式:

1. (x for x in range(5)) 列表生成式 xrange() 关键字
2. 函数中 yield 关键字 此时调用函数->创建生成器对象

多次调用->多个生成器对象间彼此独立

定义之间可以嵌套，只要不取值生成器就不干活

调用方式: next & send

相同 会在yield暂停执行并返回yield后结果

遇到了return/没有遇到yield 都会StopIteration异常

不同 语法next(<object ‘generator’>) /generator.send()

send可以传入值,且只能继续向下执行yield后语句

1. 闭包

引用-对象名,函数名赋值给一个变量 / 调用-函数名()

定义:

1.函数嵌套 2.内部函数使用了外部函数的形参或局部变量

把这个内部函数以及用到的外部函数变量称之为闭包

与函数比较:

永久存储局部变量，占用内存资源(需要手动释放)

提高代码复用性

作为实参传入 功能+数据 / 函数只传入功能

修改局部变量地址 -> nonlocal 关键字

与对象比较:

都是数据(n个属性/n个局部变量) + 功能(n个参数/一般1个内部函数)

闭包如何实现多个功能？ 返回功能元组即可

闭包更轻量级，减少内存浪费

1. 装饰器

作用:

不修改原函数的基础上，对原函数做功能扩展

- 执行时间统计

- 权限验证

- 引入日志

- 预备/清理功能

@xxx -> 语法糖 四两拨千斤 xxx可以是函数/类

Python解释器执行:

执行xxx / 将下面函数的引用当作实参传入

执行xxx结束的返回值赋值给下面函数 / 即下面函数指向x的返回值

为什么不直接调用xxx?:

用装饰器可读性更强

闭包实现装饰器:

普通闭包把外部变量视为数据 / 闭包作装饰器把外部变量视为可调用的函数

通用装饰器:

依照被装饰函数分类：有参数 / 无参数 有返回值 / 无返回值

有参数 - 装饰器装饰的函数实参传入闭包，作为闭包函数形参 \*args

有返回值 - 装饰器装饰的函数有返回值，闭包内部return

多个装饰器:

从下往上依次装饰

从上往下依次调用

装饰器带参数:

函数装饰器带参数:

执行函数并返回

返回值作为真正的执行装饰器

类装饰器带参数:

初始化一个实例对象并返回

调用\_\_call\_\_方法，返回值赋给被装饰对象

1. 拷贝

is 和 ==:

== - 判断数值是否相同

is - 判断是不是指向同一个(地址)

深拷贝/浅拷贝:

深拷贝 - 递归拷贝所有的数据 copy.deepcopy()

浅拷贝 - 顶层拷贝copy.copy() 只检测最顶层

引用 - a = b

其它方式:

切片 - 浅拷贝

字典copy方法 - 浅拷贝

拷贝数据类型:

可变类型 - 地址不同 - 拷贝顶层 / 不可变类型 - 地址相同 - 相当于引用

1. with与上下文管理器

with目的:

正确释放资源(文件/网络)

上下文管理器实现:

1. 类內定义\_\_enter\_\_() 和\_\_exit\_\_() 方法，其对象是上下文管理器
2. contextlib库里的contextManager方法
3. 内建函数

dir(\_\_builtin\_\_) 查看所有内建函数，如range

map函数:

两个参数: 可调用对象 + n个可迭代对象

按迭代顺序调用函数返回结果

filter函数:

两个参数: 可调用对象 + n个可迭代对象

按迭代顺序调用函数，if判断返回结果True则不筛，迭代返回结果

reduce函数:

from functools import reduce

两个参数: 可调用对象 + 1个可迭代对象 + n个任意对象

持续调用函数直到可迭代对象无法迭代

map/ filter/ reduce 区别:

返回值不同 map/filter 返回对象 / reduce 返回值

(不说)输入参数不同 map/filter 只接受可迭代对象 / reduce 有一个可迭代即可

运算机制不同 map/filter 只记录迭代顺序 / reduce 会把当前迭代值作为下一次输入

1. Python多任务

主要目的 提升程序的效率 缩短运行时间

1. 时间片调度

时间片调度 - 操作系统为了让多个程序都能得到执行的机会，给每个程序分配微量的执行时间，微观上看单核一直在执行一任务，宏观来看因为时间间隔很短，单核在同时执行很多任务

1. 并发/并行
   1. 并发: 单个核心同时执行多个任务
   2. 并行: 真正的多任务，多个核心执行多个任务
2. 线程 - 操作系统最小执行单元
   1. 主线程: 程序运行时默认有一个主线程
   2. 共享数据: 全局变量
   3. 多线程: threading库调用

语法 - Thread()创建线程对象 target参数传入函数引用 args参数传入可迭代参数类型(元组) kwargs参数指定形参名传入(字典) threading.enumerate()查看线程

thread方法start()执行线程 join()主线程等待子线程执行

setDeamon()守护线程 主线程执行完毕线程销毁

setName() 设置线程名字 getName() 获取线程名字

用类创建线程 class中继承Thread包含run()方法

理解1: 伪多线程(因为GIL锁) 同一时间只能进行一个操作

2: 底层 真正的进程线程管理由操作系统实现 Python只是利用的它的接口

3: 主线程会等待子线程结束后做垃圾回收

4: 多线程的执行顺序由操作系统决定

1. 线程池 - 不是说线程开的越多，效率越高
   1. 语法 concurrent.futures模块下的ThreadPoolExecutor类
      1. pool = ThreadPoolExecutor(100) 创建了100个线程
      2. pool.submit(函数,参数) 往线程池里安排任务 / 有空余线程执行，否则等待 / 返回一个对象future 执行future.add\_done\_callback(函数) 区分任务
      3. pool.shutdown() 等待线程池所有任务执行完毕 类似join()
2. 线程安全 -

多线程处理同一个数据的时候，可能会造成数据混乱，所以通过锁来保证线程安全

锁 lock = threading.Rlock() 对象

* + - 1. 申请锁 lock.acquire() 申请成功->加锁 / lock.release() 释放锁 / 只有锁被释放后才能继续申请
      2. 也可以用上下文机制 with lock 自动执行acquire() 和release()
      3. 线程安全的操作 - 列表append, 切片, pop等 / 字典赋值

Lock和RLock:

Lock同步锁 一个线程只能锁一次 可以在其他线程释放

RLock递归锁(解铃还需系铃人) 可以多次acquire和release 但必须在同一个线程下执行

Lock 同步锁不适合函数嵌套的时候线程安全保证，此时应该用RLock

死锁 - 由于资源竞争

一条线程同步锁锁了很多次 / 两个线程使用对方已经申请的锁

1. 进程 - 操作系统分配调度资源的基本单位
   1. 语法
      1. multiprocessing库
      2. multiprocessing.Process类 / 参数和多线程相同 count\_cpu()方法 获取cpu个数
      3. start() 方法 join() 方法 和线程一样
      4. .deamon属性 守护进程 子进程在主进程结束后自动关闭
      5. .name属性 设置名字
      6. os.getpid() 获取进程id os.getppid() 获取父进程
      7. 自定义进程类 继承Process类 类內编写run方法
      8. 理解: windows系统下多进程不在 if \_\_name\_\_ == “\_\_main\_\_”启动会报错，因为创建进程 linux 基于fork (不报错) / windows,mac 基于spawn(报错)
   2. 三大机制 multiprocessing.set\_start\_method(“fork”/”spawn”)
      * 1. fork - 拷贝主进程的所有资源
        2. spawn - 创建一个python解释器，去运行新的子进程，只生成必备的资源，各自维护各自的数据
           1. 可以以参数形式传递一些对象，例如数组、字符串
           2. 一些特殊的对象如类的实例，锁，必须要在子进程自行创建，无法传参
        3. fork拷贝已申请的锁时，拷贝到子进程的主线程
   3. 进程通信
      1. 基于multiprocessing库里的Value和Array数据共享（基本不用） 用了一些c的知识 Value(‘i’, 666) 了解即可
      2. 基于multiprocessing库里的Manager类和上下文管理器
      3. 基于multiprocessing库里的Queue类，FIFO，put get
      4. 基于multiprocessing库里的Pipes类 par\_con, child\_con
         1. 特点 双向传输 / 类似于通信机制
         2. 语法 send() 发送 recv() 等待接收
   4. 进程锁
      1. 目的: 防止资源竞争，多个进程同时抢占一个资源
      2. multiprocessing库里的 RLock
         1. 方法 acquire申请 release释放
      3. 进程锁和线程锁的区别:
         1. 进程锁可以传参传入，线程锁不行
      4. spawn模式下需要等待子进程运行完毕再结束主进程
   5. 进程池
      1. 目的: 防止无限制的开辟进程，导致程序效率变慢
      2. concurrent库里的ProcessPoolExecutor
         1. 语法，调度的机制类似于线程池 都是有空余进程/线程就执行任务
         2. pool.shutdown(True) 进程池的关闭
      3. 和线程池的区别:
         1. 进程池的add\_done\_callback() 由主进程执行 / 线程池的add\_done\_callback() 由线程池里的子线程执行
2. 进程和线程的区别

进程是程序运行时，操作系统（工厂）开辟的单元，调度和分配硬件资源；线程是真正工作的单位（工人），进程（车间）是为线程提供资源的单位

一个进程中至少有一个线程

多进程比多线程开销大:

- 如果想要利用计算机的多核优势，适合多进程，比如计算密集型（大量数据计算）

- 如果不需要，适合多线程，比如IO密集型（文件读取）

1. GIL锁 全局解释器锁 CPython中特有
   1. 一个进程中 / 同一时刻只有一个线程被CPU调度
   2. 计数器 / 有耗时操作释放GIL锁
   3. 为什么有了GIL还需要锁？ 锁的对象不一样
      1. GIL防止多个线程抢CPU / 防止资源竞争
      2. 锁主要是防止多个线程同时修改某个全局变量 / 防止资源竞争
   4. 如何解决 -> 换语言 / 换解释器
2. 垃圾回收机制
   1. 小整数对象池:
      1. Python解释器为了提高效率，通用数据在程序开始已经创建，只等待被使用比如 int [-5, 256] 创建它们的引用地址相同
      2. 大整数对象池 - 超出上面范围 - 节约内存
   2. Intern机制：（字符串驻留）
      1. 引用相同的普通字符串地址相同
      2. 默认自动开启，对特殊字符串(空格等)无效
   3. 垃圾回收: Garbage Collection
      1. 引用计数为主，分代收集为辅 - Python解释器管理
      2. C使用malloc申请，free释放，没有正确释放则会产生垃圾和内存泄漏
      3. Python尽量删除垃圾
      4. Python为了知道每个对象有多少个变量指向他，会存储对象的引用计数
         1. 例如 obj = Class\_a() 引用计数1 / obj2 = obj 引用计数2
      5. 优缺点
         1. 简单
         2. 实时性 引用为零则释放 不等待 -> 收集垃圾的时间均摊
         3. 维护引用计数消耗资源，使得Python运行慢
         4. 循环引用
      6. 循环引用
         1. 对象a和对象b在彼此属性里相互引用 a.xx = b / b.xx = a 此时删除a和b对象的时候，两者的引用计数为1，故不会被GC释放
      7. 隔代收集： 解决循环引用
         1. 把所有的对象存储在链上，一共三代，默认存储在一代
         2. 清理机制，即把所有对象的引用计数减一 减一后计数等于零，释放 剩下的移动到下一代链上
         3. 触发机制 gc.get\_threshold()
            1. 一代 由GC阈值决定何时运行，比如垃圾对象，也就是创建对象数减去释放的对象数达到阈值
            2. 二代 一代链触发检测机制数达到一定阈值
            3. 目的就是为了提高效率，因为每一次检测清理都会消耗大量时间
   4. 语法 gc 模块 gc.disable() 关闭gc.collect() 手动调用垃圾回收
   5. 引用计数:
      1. 加一的情况
         1. 创建对象
         2. 引用对象
         3. 作为实参传入函数
         4. 存储在容器中 列表/元组
      2. 减一的情况
         1. 显式销毁 del
         2. 变量引用新的对象，旧的 -1
         3. 离开作用域 如函数结束执行
         4. 从容器中删除 或者容器被销毁
      3. 查看引用计数
         1. sys库getrefcount()
3. Python数据结构
4. 列表

底层实现 - 顺序表

存储方式 - 顺序存储，列表对象指向列表第一个元素的内存地址

兼容不同类型的对象，因此连续存放的不是数据，而是指向数据的地址

1. Python队列 queue库
   1. 堆栈queue.Lifoqueue() LIFO
   2. 队列queue.Quque() put() get() FIFO / 堵塞
   3. 优先级队列 queue.PriorityQueue() 元组 / 优先级(越小优先级越高)+存放数据
2. Python字典的底层实现 - 哈希表