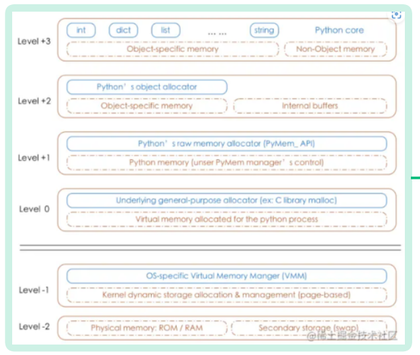
python高频面经

参考文章
- https://blog.csdn.net/qq7835144/article/details/117294737
- 进程线程：https://blog.csdn.net/qq_62789540/article/details/123205717
- 详细的元组原理：https://zhuanlan.zhihu.com/p/361939906?utm_id=0

垃圾回收
- 引用计数
  - 指被其他对象引用的次数，当计数为0时进行销毁。
  - ● 引用计数增加的情况：
    1. 一个对象被分配给一个新的名字（例如：a=[1, 2]）
    2. 将其放入一个容器中（如列表、元组或字典）（例如：c.append(a)）
    ● 引用计数减少的情况：
    1. 使用del语句对对象别名显式的销毁(例如：del b)
    2. 对象所在的容器被销毁或从容器中删除对象（例如：del c ）
    3. 引用超出作用域或被重新赋值（例如：a=[3, 4]）
- 标记清除
  - 主要使用在list，dict等容易造成循环引用的位置，用于针对引用计数无法解决的循环引用的问题。遍历对象，如果当前对象是可到的我们进行标记，不可达对象则清除。同时如果有循环引用，会把循环引用的对象引用-1。
- 分代回收
  - 分为年轻代，中年代，老年代。我们认为存留越久的数据被使用的概率就越大，所以更应该保留。当某一代的内存创建和销毁到达了一定比例时就会触发当前代和更年轻代的GC

内存管理
每次申请内存会导致存在非常多的内存碎片，使用内存池可以一次性申请一串连续的地址空间，减少内存碎片的产生。
level 3：用于存放python内置的类型
level 2：用于存放内存小于256kb的申请
level 1：用于存放内存大于256kb的申请，内部是使用C的malloc等方法



GLI
在CPython中，由于全局解释器锁，在同一时刻只能有一个线程进入解释器，只能有一个线程执行Python代码(即使某些面向性能的库可能会克服这个限制)。如果希望应用程序更好地利用多核计算机的计算资源，建议使用多进程。但是，如果您希望同时运行多个I/O密集型的任务，线程仍然是一个合适的选择。

进程之间的通讯方式
- 管道
  - 管道这种通信方式效率低， 不适合进程间频繁地交换数据。
    对于匿名管道，它的通信范围是存在父子关系的进程。
    对于命名管道， 它可以在不相关的进程间也能相互通信。
- 消息队列
  - 消息队列声明周期跟随系统内核，如果没有释放消息队列或者关闭操作系统，消息队列会一直存在。
- 共享内存
  - 共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中。
- 信号量
  - 信号量其实是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的数据。也就是常说的PV操作，P减一，V加一。
- 信号
  - 信号一般用于一些异常情况下的进程间通信，是一种异步通信，它的数据结构一般就是一个数字。
- socket
  - 不同主机之间的进程通信
- 总结

IPC 机制	数据抽象	参与者	方向	内核实现
管道	字节流	两个进程	单向	通常以 FIFO 的缓冲区来管理数据。有匿名管道和命名管道两类主要实现
消息队列	消息	多进程	单向 双向	队列的组织方式。通过文件的权限来管理对队列的访问
信号量	计数器	多进程	单向 双向	内核维护共享计数器。通过文件的权限来管理对计数器的访问
共享内存	内存区间	多进程	单向 双向	内核维护共享的内存区间。通过文件的权限来管理对共享内存的访问
信号	事件编号	多进程	单向	为线程 / 进程维护信号等待队列。通过用户 / 组等的权限来管理信号的操作
套接字	数据报文	两个进程	单向 双向	有基于 IP/ 端口和基于文件路径的寻址方式。利用网络栈来管理通信

封装继承多态
面向对象的思想是把一个项目、一件事情分成更小的项目，或者说分成一个个更小的部分，每一部分负责什么方面的功能，最后再由这些部分组合而成为一个整体。
- 封装
  - 将底层实现屏蔽，只对外暴露使用方法，保护数据。
  - 隐藏信息，保护数据，降低耦合
- 继承
  - 子类继承父类，减少相同的方法变量的重复实现
  - 提高代码的重复性，为多态提供条件
- 多态
  - 多态是指在父类中定义的属性和方法被子类继承之后，可以具有不同的数据类型或表现出不同的行为，这使得同一个属性或方法在父类及其各个子类中具有不同的含义。

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)。
- is：比较对象间的唯一身份标识id是否相同
- ==：比较value值是否相同

深浅拷贝
- 复制
  - 仅仅提供一个指向原来对象的指针
- 浅拷贝
  - 拷贝的是对象，但是对象内如果存在引用还是取的源数据
- 深拷贝
  - 针对引用的引用同样重新创建一个全新的对象

线程&进程
- 进程
  - 进程是系统进行资源分配和调度的基本单位
  - 由程序，数据集合和进程控制块三部分组成。
  - 由于进程比较重量，占据独立的内存，所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。
    运行程序最少会需要一个进程，并且单核cpu一次最多运行一个进程。进程一般使用pid来进行唯一标识
  - 通常使用multiprocessing来实现
  - 多进程多用于CPU密集型任务，例如：排序、计算，都是消耗CPU的。
- 线程
  - 线程是系统调度的最小单元。
  - 它可与同属一个进程的其他的线程共享进程所拥有的全部资源。线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。
    单核cpu一次最多运行一个线程，一个进程的线程可以分布在多个Cpu中，所以只有多核才能实现多个线程同时运行。
  - 通常使用Thread类中的threading
    常用方法：
    Thread.run(self) # 线程启动时运行的方法，由该方法调用 target 参数所指定的函数
    Thread.start(self) # 启动线程，start 方法就是去调用 run 方法
    Thread.terminate(self) # 强制终止线程
    Thread.join(self, timeout) # 阻塞调用，主线程进行等待
    Thread.setDaemon(self, daemonic) # 将子线程设置为守护线程
    Thread.getName(self, name) # 获取线程名称
    Thread.setName(self, name) # 设置线程名称
  - 多线程多用于处理IO密集型任务频繁写入读出，cpu负责调度，消耗的是磁盘空间。
  - 在大多数编程语言中因为切换消耗的资源更少，多线程比多进程效率更高
- 池并发
  - concurrent.futures 模块中的 Executor，Executor提供了两个子类，即 ThreadPoolExecutor 和 ProcessPoolExecutor，其中 ThreadPoolExecutor 用于创建线程池，而 ProcessPoolExecutor 用于创建进程池
    如果使用池并发来管理并发，只需要将task函数交给池即可。

常用数据类型的底层原理
- dict原理
  - python3.6及之前
    - 字典底层是维护一张哈希表，哈希表中的每一个元素又存储了哈希值（hash）、键（key）、值（value）3个元素。
      entity = [hash, key, value]
    - 1.计算hash，和mask做运算，得到将要存放的坐标index
      2.如果被占用，比较key值，如果相同更新value
      3.如果不同向下寻找空位置，直到找到为止。
  - python3.7及之后
    - 字典维护一个之前的哈希表,同时新增一个indices表辅助，针对稀疏矩阵内存上有优化，因为entity没有那么多空位置了
      indices = [index, None, None, index]
      entity = [hash, key, value]
    - 1.计算hash，可mask做与操作，计算出index，得到需要插入的indices的下标位置
      2.找到indices的对应位置，存入len(entity),代表了哈希表的索引，映射关系
      3.如果出现冲突同之前版本
- list原理
  - 使用连续存储空间。
    使用功能分离式结构，即表头和元素内容分离，更改时始终指向的地址不变。
    元素可以是任意类型，则使用的是元素外置类型，存储的是对象的引用。
    扩展策略：空list申请8个元素大小的内存空间，如果满了4倍扩容，到达50k之后2倍扩容。
- set原理
  - 底层使用hash实现
  - 插入：同hash
  - 删除：如果找到了hash，比较值是否相等否则继续找。
  - 去重：set的去重是通过两个函数__hash__和__eq__结合实现的。
  - 扩容：当剩余值小于三分之一时，双倍，重新排放元素。
- tuple原理
  - 基本结构：
    typedef struct {
    struct _object *_ob_next;
    struct _object *_ob_prev; // 双向环状链表中上一个和下一个，Python内部将对象放到链表中便于进行内存管理。
    Py_ssize_t ob_refcnt; // 引用计数器，即：有多少变量使用了这个列表对象。
    Py_ssize_t ob_size; // 元素个数
    PyObject *ob_item[1]; // 存储元组中的元素 [指针,]
    } PyTupleObject;

常用的数据类型
- 不可变
  - 数字（Number）
    - 整型（int）、浮点型（float）、复数（complex）、bool
  - 字符串（String）
  - 元组（Tuple）
    - 但是元组的元素值不可修改也不能删除，可以进行分片和连接。只能向后添加或者连接
- 可变
  - 列表（List）
  - 集合（Set）
  - 字典（Dictionary）

生成器&迭代器
- 迭代器
  - 可以被next函数调用并不断返回下一个值的对象被称为迭代器(iterator)
  - 可迭代对象不一定是迭代器（list），迭代器一定是可迭代对象
  - 只有在需要的时候才会去调用next方法，是惰性计算。只能取下一个，并且只能取尽否则不能知道长度。
  - 底层：迭代器中会有__iter__和__next__魔法方法，__iter__返回自己，使用__next__来返回下一个对象。
- 生成器
  - 其本质上可以理解为一个特殊的迭代器，生成器具有和迭代器一样的特性。但是它们在实现方式上不一样。我们可以通过两种方式创建生成器：生成器表达式，生成器函数。
    生成器表达式：b = (i for i in range(5))
    生成器函数：yield n