Python

Data Structures: https://docs.python.org/3/tutorial/datastructures.html#sets

Operators	*Floor Division* division without decimal is *//*, division with decimal is */* . Example: 1 / 2 = 0.5, 1 // 2 = 0 *Bit Manipulation* a&b, a|b, a^b(Exclusive OR), ~a(NOT), 1<<bit(left shift)
Constants	Max/min Int: sys.maxsize, -sys.maxsize

Switch	`match x:`
	`  case "hi":`
	`  print(x)`
	`  case "hey":`
	`  print(x)`
	`  case _:`
	`  print("not matched")`

Enumerate an object(string, list, map…)	`for index, value in enumerate(object):`

Lambda function	`Lambda:`
	`add_func = lambda a, b : a + b`
	`add_func(5,6)`
	
	`map:`
	`list = [1,2,3]`
	`incremented_list = map(lambda x: x+1, list) # [2,3,4]`
	
	`filter:`
	`a = [0,1,2]`
	`filtered_a = list(filter(lambda x: x>0, a)) # [1,2]`
	
	`dedup:`
	`l = [1,1,2]`
	`deduped = list(set(l)) # [1,2]`

Optional	*number: Optional[int]* means that the number can be an integer or *None*(python's null)

List node	`class ListNode:`
	`  def __init__(self, val=0, next=None):`
	`  self.val = val`
	`  self.next = next`
	
	注意:costructor的default param不能是mutable object(list etc)。否则该default param会被所有实例反复修改。如下
	`class ListNode:`
	`  def __init__(self, children=[]):`
	`  self.children =children`
	`l1 = ListNode()`
	`l1.children.append(1)`
	
	`l2 = ListNode()`
	`print(l2.children) # 包括1！`

Create a 2D array	`matrix = [[0 for _ in range(col)] for _ in range(row)]` 或者 `[[0]*col for _ in range(row) ]` 不能用`[[0]*col]*row`, 因为python会把同一个数组指针复制row次，而不是row个不同数组

(De)Queue	`from collections import deque`
	
	`queue = deque([1,2,3])`
	`queue.append(4)`
	`queue.extend([4,5,6])`
	`queue.popleft() # pop first element`
	`len(queue)`
	`queue[0] # peek first element`
	`queue.pop() # pop last element`

Heap/Priority Queue	`# Only support min heap. For max heap, add negative signal`
	`import heapq`
	`data = [2,3,5,1]`
	`heapq.heapify(data)`
	
	`data[0] # min value`
	`heapq.heappush(data,6)`
	`print(heapq.heappop(data))`

<table>
<tr><td></td><td>Custom sorting:
make element a tuple, sort on the first tuple item
If the element itself is not comparable(e.g. a custom struct like node), that will cause error. For that, make the element's id the second item in tuple.</td></tr>
<tr><td>Array/Str Indexing</td><td><pre>array[-1] # last element
array [start:end:step] # a subslice of array
array.reverse() # reversed array
array[:3] # first 3 elements</pre></td></tr>
<tr><td>Sorting</td><td><pre>array.sort(key=lambda element: (element[1],element[2]), reverse=True) #多个sorting key</pre></td></tr>
<tr><td>Dict/Set</td><td>Dict
<pre>dict = {}
dict['key'] = 1
del dict['key']
for key in dict:
for value in dict.values()
for key, value in dict.items()</pre>Defaultdict
<pre>from collections import defaultdict
d = defaultdict(int)
print (d['invalidKey']) # print 0 as int is set as default</pre>
<pre>Set:
set = set([0,1]) # note that you CANNOT use {} to initialize a set!
set.add(1)
set.discard(1)
set1 | set2 # union
set1 & set2 # intersection</pre></td></tr>
<tr><td>Multiple return values in function</td><td>Def func(.....) -> Tuple[bool, bool, bool]:</td></tr>
<tr><td>String Manipulation</td><td><pre>Join:
array = ["aa", "bb"]
separator = ","
separator.join(array)

Split:
s.split() # split by space. consecutive tab etc
s.split(" ") # strictly split by single space, e.g. 'a b' -> 'a', ' ', 'b'

Strip
s.strip('abc') # remove all a,b,c</pre></td></tr>
<tr><td>Pass by Reference Value</td><td><pre>Python 是 pass by assignment. 如果re-assign value, 那么并不是改原指针指向的值 而是给一个新的指针; 如果在原指针直接改 那么会改变值

val = [1,2,3]
def reassign(val):
    val = [1,2,3,4]
reassign(val)
print (val) # doesn't change

def append(val):
    val.append(4)
append(val)
print (val) # change</pre></td></tr>
<tr><td>Copy</td><td><pre>import copy
set = set([1,2)
list = [[1],[2]]
print(copy.deepcopy(set))</pre></td></tr>
</table>

Algorithms

Common Sorting Algorithms	

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

排序算法分析.jpeg

归并排序
- 先对前半和后半部分递归，最后归并两个有序部分
- 空间
  - 数组：是O(n)因为归并部分需要复制值。不是O(nlogn)因为最后归并的时候递归部分的空间已经释放了
  - 链表：是O(logn)因为归并部分不需要新的空间。
- 稳定

快速排序
- 任意选一个pivot然后根据pivot排序(一般是第一个元素）。注意**pivot**应该单独取出而不是放在 递归里，否则如分裂出一个空的子集和一个非空子集，那么非空子集没有变小，会陷入死循环。
  - 数组：维护前后两个指针指向最小区末尾和最大区前面，每个循环不停自增/自减两个指针，直到找到第一个大数和最后一个小数，然后交换
  - 链表：直接开大/小两个表头，然后加节点，合并两个表
- 空间就是递归层数，最好情况O(logn)，最坏O(n)
- 同理，时间最好O(nlogn),最坏O(n^2) 但是总体是最快的
- 不稳定，因为交换的时候越靠前的元素会交换到越靠后

堆排序
- 不稳定，因为Pop的过程会把后面的元素放到前面
- 只用于数组
- 过程
  - Push: 先加到数组尾，然后sift up和父亲交换，直到小于父亲。父亲是(i-1)//2
  - Pop:把数组尾放到根，然后sift down和最大大的孩子交换，直到小于孩子或者。孩子是(2*i+1, 2*i+2)
  - 建堆: 从空开始，逐个push新的元素。如果已经非空，则逆序把非叶节点sift down，因为轮到某节点的时候，其两个不稳定子树都已经建堆完毕。

非比较排序：都是稳定，都不是原地算法

基数排序
- 先按照个位排入桶，再按照十位入桶，此时因为已经按个位排好序，所以若十位相同则自动按个位排序，依次往复
- 额外空间为O(n+桶数），需要额外空间；时间是O(n+位数）

桶排序
- 设置M个桶(区间），把元素放到桶内。每个桶内部用其他算法排序
- 额外空间是O(n+桶数），一般桶数要比较大一点；时间理论上是常数，如果桶够多。
- 需要数据分布非常均匀，不然所有数都进同一个桶等于没排序。用得不多

计数排序
- 桶排序的进阶版。需要数值是有限集并且范围很小（如1-10的整数）
- 先走一遍计算每个值出现次数，再从1-10逐个打印，出现几次打印几次。
- 额外空间是O(桶数）

综合比较
- 对于数组用快排，因为空间优势，不用调用新的空间，并且因为不调用新空间快排更快。
- 对于链表用归并，因为链表的归并调用空间跟快排一样，并且归并保证每次均匀分配所以会更快。
- 堆排序一般不用于排序，而是Top K问题等
- 非比较排序要么应用场合有限（需要特殊条件如分布均匀或者有基数）要么需要额外空间，用得不多

Heap/Priority Queue	

- 一般用于求Top K/Median问题。维护大小为K的堆，堆顶就是第K大的。
- 只支持删除/查找堆顶。

Union Find	

- 用处
  - 判断两个元素是否属于同一集合(如两点互相联通）
  - 数一个图里的集合个数
- 基本结构
  - 一般用数组表示。A[i]表示第i个元素的父亲。初始化是自己
  - union合并两个任意元素所在集合
  - find找到元素的父亲

有两种实现 :https://www.jianshu.com/p/8d3e9bbe135a
1. Quick Find. 线性结构。 每次union的时候 把被改节点所有孩子的父亲都更新 O(1) Find, O(n) union. 建树O(n^2)
2. Quick Union 树结构（更常用）。
   - 通过平衡树的方法，Find/Union 都是 O(logn)
   - 加上路径压缩（每次find的时候顺便把经过节点的父亲改成根节点），则Union/Find 接近O(1)，但还不是线性

Tree	

Traversal
All use recursion or stack with the current node.
解法**1**
InOrder: left, mid, right. 每个节点进出栈一次 出栈时左孩子已经打印过了

PreOrder: mid,left,right; 同 inOrder
PostOrder: left, right, mid. 因为一个节点要push两次，第一次遍历左边 第二次遍历右边 需要记录前一个打印的节点 如果是当前的右孩子那么打印 如果不是那么继续push

解法**2**
把所有孩子反序入栈，前序和后序都可以用，N-ary也可以用 不需要当前节点指针 注意这样的话栈并不等 于路径因为兄弟节点也在

解法***3 Morris Traversal***
前中后序都可以用 常数空间 线性时间 需要临时改变树的结构 最后复原
原理：
前中序，第一次经过一个节点时 把其左孩子的最右边孩子的右端连上该节点 然后进入其左孩子 左孩子遍历 到最后一定是最右边孩子 这样最右边孩子可以直接进入原节点
后序，遍历顺序一样 但最右边孩子进入原节点时 不是打印原节点 而是倒序打印原节点左孩子到最其右孩子路径上的节点 并且要给根节点加一个虚拟节点作为父亲

(De)Serialize a Tree
- Serialize: DFS with parenthesis, e.g. [1[2][3]]表示1有两个孩子2和3
- DeSerialize: 同样DFS，维护一个全局索引指向当前字符，DFS里每次都从开括号开始，先建立节点，然后把索引指向后面，不停递归孩子直到索引指向闭括号。最后返回当前节点

Binary Search Tree(BST)
一般的套路
- 需要遍历所有节点：中序遍历。很少前序或者后序。
- 根据特性需要实现O(h)查找: 比较大小
- 插入，查找都很简单。难的是删除。递归实现。对于某个节点要删除某个值：
  - 如果值小于/大于该节点值，则递归左/右子树
  - 如果值等于该节点
    - 若该节点左右子树都是空，则删除该节点（需要传父节点并更新）
    - 否则，找到节点右子树的最左孩子（右子树最小值）将其值赋给当前节点。然后把该孩子递归删除。若右子树为空则左子树类似操作。

Trie
假设有N个字符串，每个长均为L，则
- 实现插入，查整字符串，查前缀操作，时间都是O(L)
- 使用场景
  - 从第一个字符开始，逐个字符进行遍历查找（如"a","ab","abc")，则查找时间缩短L倍
  - 查找前缀
- 比HashMap的优势
  - 可以查询某个前缀是否存在
  - 省空间。空间最差是O(NL)，但最好是O(N)，当所有字符串都有同样前缀

实现注意的点
- 直接用一个dict{}代表Trie Node，不用另外定义
- 用'$'的key代表isWord
- 参考

Graph

DFS-Backtracking
算法
维持一个visited set/matrix, 每到一个节点
1. 先检查节点是否出边界(Null/index out of bound)
2. 节点是否visited(树可省略)
3. 节点是否找到答案(terminal), 若是则直接返回找到
4. 节点设置为visited(树可省略)
5. 对每个邻居 判断是否递归遍历
6. 节点设置为unvisited,返回未找到(树可省略)
7. 注意只有后序遍历能打印*path!*

Memorization
对每个节点保存状态 避免反复DFS。

时间复杂度
假设DFS/BFS的depth是L, 每个节点有N个邻居 时间复杂度是L^N

BFS
找最小路径:
- 先判定初始节点是否是终点/符合入队条件，若是则入队和加入访问集，步数为0
- 每一步的一开始步数自增
- 记录队列当前长度，出队所有元素
- 对每个元素找他的邻居
  - 如果邻居是终点，直接返回步数
  - 如果邻居已经访问，跳过
  - 把邻居加入访问集，入队。注意不能先入队再看是否加入访问集，会造成多次入队

DFS/BFS 比较
- BFS: 求最短路径
- DFS: 求所有路径
- 空间复杂度也重要。
  - 完全二叉树的问题，DFS (*logn*),BFS (*n*)，用DFS优先
  - 矩阵问题，DFS(n*n), BFS一般(n)因为只包括边缘，用BFS优先

Dijkstra
对一个有权图，求所有节点到一个起点的最短距离（Google Map)
步骤
- 对每个顶点维护到起点已知最短距离，初始化无穷大
- 维护一个已访问集合装所有已经求得最小距离的顶点，初始化只包含起点
- 每一步：
  - 取出不在已访问集合里已知最短距离最小的顶点，将它加入已访问集合，其已知最短距离就是实际最短距离
  - 对该顶点所有邻居更新他们的已知最短距离

	○ 反复一直到所有顶点入队 证明 ● 假设对前N个加入的顶点成立 ● 对第N+1个顶点A，其已知最短距离在未入队顶点最小 ● 假设有一条路径是最短路径 1. 如果该最短路径路上所有顶点都在队内，则路径距离肯定已经计算到A已知最短距离 2. 如果该路径有顶点不在队内，那对于路径上第一个不在队内的顶点B来说， ■ 该路径B到起点的距离已经计算到他当前最短距离 ■ 则B当前最短距离<路径上B到起点距离<路径上A到起点距离 ■ 如果路径上A到起点距离小于当前最短距离，得到B当前最短距离<A当前最短距离，矛盾。 ■ 所以只可能是情况1，A当前最短距离就是实际最短距离 复杂度 ● 如果用数组维护未入队顶点的当前最短距离，则时间复杂度O(V*V)，因为在每两个节点联通时每一步都是O(V),空间复杂度除去图本身是O(V) ● 用堆维护的话，则时间复杂度O(E*logV)，是每次加新节点时把其邻居的最新的最短距离入堆的时间（最后堆里可能有E个元素），空间复杂度是O(E+V) ● 空间复杂度除去图本身都是O(V) **Topological Sort a directed graph(Cycle Detection)** *BFS方法* 一个queue存所有indegree = 0 的节点，存所有节点的indegre和outgoing edges 一直dequeue节点 去除他的outgoing edge再看看有没有新的indegree = 0的节点 DFS方法 每个节点三个状态: unvisited, visiting(遍历路径上), visited(结束遍历)、 DFS所有节点, 对每个节点先标成visiting 再DFS邻居，最后标成visited。 途中若碰到visiting则有环，若碰到visited则说明该节点已经访问过，跳过。 **Bi-Directional BFS** ● 维持两个队列/访问集，一个起点一个终点 ● 每次选小的队列，把所有元素出队。这样保证队列存的是一整层，也保证路径最短 ● 出队的元素，检查是否在对面的访问集，是的话结束遍历 无向联通图找到所有环里的边 算法 从任何节点开始DFS前序遍历。对每个节点维护 1. 他的遍历顺序 2. 他访问邻居的最小节点顺序(默认值是自己的顺序） 每个DFS中，对于一个现节点和他的每个邻居 1. 如果是父亲，略过(这个所有无向图都要注意) 2. 如果遍历顺序为空，说明没遍历过，递归遍历(现节点顺序+1) 3. 如果邻居的访问最小节点顺序大于现节点顺序，说明现节点和邻居的边不在环里。否则在环里 4. 最后更新现节点的邻居最小节点顺序 算法注释 1. 现节点顺序肯定大于邻居，否则邻居比现节点先遍历，在之前遍历的时候肯定会接触现节点，矛盾
Dynamic Programming	● 动态规划有两种实现： ○ 多重循环，思考有难度但是最优，优先用这个。有两种都可以尝试 ■ Top-down，最容易想到，从开端（例如数组头部）开始 ■ Bottom-up，从结果（例如数组尾部）开始反推 ○ 记忆化搜索(Memorization)，DFS+存重复运算结果， ■ 比纯DFS好但是仍然要递归并且难分析复杂度，一般尽量避免 ■ 使用场景是依赖关系跟自然顺序无关(如上到下/左到右，例子 Longest Increasing Path in Matrix)或者依赖方程跟路径有关，这个情况下用Memorization代码会更易懂。 ● 用动态规划求解的题目特点： ○ 返回 最大最小值/判断可行/方案个数，不求所有具体方案 ○ 暴力算法复杂度是输入长度的指数级别（指数一定动规，非指数也有可能动规） ○ 输入是序列而非集合（这个不绝对但适用大部分情况） ● 常用技巧/注意： ○ 以空间换时间。对于反复计算的数据，用较大空间把他存起来，这样空间增加时间减少 ○ 如果依赖关系有环，想办法拆环，比如拆成两个依赖关系没环的子序列 ○ 滚动数组。如果状态只依赖于前一个状态，则不用存所有状态，而是定义当前和之前的状态滚动刷新。 ■ 一维：用两个单一变量 ■ 二维：用两个数组 不同动态规划类型，定义状态 ● 坐标型。给一个矩阵/数组，起点终点，行走规则 ○ 状态：F[i]表示起点到i状态的目标值。依赖于F[i-1], F[i-2]...F[i]里的一到多个 ○ 状态可以是物理坐标，也可以是步数，高度等限制，依赖方程里不能有环 ● 单序列型。给一个字符串/数组。 ○ 状态：F[i]表示前i个元素（如prefix)的目标值。 ○ 长度为N，则设置N+1个状态，F[0]单独做初始化 ● 区间型。给了一个字符串/数组。 ○ 状态：F[i][j]表示在[i,j]区间的目标值。依赖于区间长度(求最长/最短)或区间起点/终点 ● 双序列型。给两个字符串/数组。 ○ 状态：F[i][j]表示第一个序列前i个和第二个序列前j个子序列的目标值 ● 策略型。两个人玩游戏（如取硬币），轮流做操作（如取一个或取两个） 先达到一个目标（如100元）就赢，问先手能否赢。 ○ 状态：跟目标相关（如钱数）。如果任何一个选项（选任何一个硬币）使得对手输，则我赢。反之我输。
Linked List	*Dummy Head* ● 给链表假表头，使用的情况是返回的表头可能跟原表头不一样，比如 ○ remove duplicate in sorted list. 如果表头是duplicate，那么表头本身可能被remove ○ merge 2 sorted linked list. ● 不需要清除假表头，因为他没有referece自动会清除 常见操作

<table>
<tr><td></td><td>

删除现节点后面的Node: cur.next = cur.next.next
现节点后面插入一个Node:

new.next = cur.next
cur.next = new

把一个Node从原来位置插入新的位置:先删除，再插入
找表的中间：注意要找中间前一个节点， 方便把中间节点从原链表分离（如Merge Sort)

midPrev, fast = head, head
while fast and fast.next and fast.next.next:

fast = fast.next.next
midPrev = midPrev.next

表长为2，则midPrev指向第一个元素，表长为3则也指向第一个。表长为1应该直接返回

合并两个链表

while ptr1 and ptr2: (add ptr1 or/and ptr2)
append ptr1 or ptr2 to current ptr if not null, since at most one of them is non null

双指针

快慢指针用来链表找环:快指针比满指针快一倍。一直到快慢指针相遇。 LC141
如果要返回环开始的地方:则一个指针放到链表头，另一个放到相遇点，匀速一起前进，直到相遇。原理是链表头到环起点的距离与相遇点到环起点的距离相差环的长度的倍数。LC 142
去除离终点距离为N的节: 双指针设置距离为N，一起前进直到前面的指针到列尾 LC 19
Partition/Sorting题目：不用交换，可以开新链表然后插入。 LC 86

数组交换是为了in-place，但是链表的Node本来就可以移动，不会用到额外空间。
数组交换也是为了保存数组在内存里的连续性，但链表的Node在内存中本来就是离散的。

倒转链表问题

判定链表非空
初始化为prev = None, cur = head
每次 next = cur.next, cur.next=prev, cur,prev = next,cur
最后的prev就是新的表头，因为cur已经空了

双链表

每个节点有prev, next, val
用于LRU, LFU
双链表本身维护Dummy Head, Dummy Tail. 支持以下功能

从头删除（删除最老的值）
从尾加入(加入最新的值）
删除任意指定的节点（保证在链表里）。用于更新。

</td></tr>
<tr><td>Monotone Stack</td><td>
单调（递增或递减）栈用来处理一维数组。

用法: 遍历数组，对每个元素，先出栈所有比该元素大的，然后入栈该元素。
特性(以递增栈为例）：

元素入栈时的栈顶（也是其出栈后的栈顶）是其左边第一个比他小的元素。用反证法。假设入栈A时栈顶不是其左边第一个小的元素，那么左边第一小的元素B肯定已经出栈了，则在B与A之间存在另一个比B小（从而比A也小)的元素使得B出栈，那么B不是左边第一小的元素，矛盾。
导致元素A出栈的当前元素B是其右边第一个比他小的元素 。否则在遍历B之前A肯定已经出栈。
栈内是否允许相同值由以上两个特性决定。若求左/右第一个小的元素则允许。因为栈顶出栈的时候当前元素需要小于他，所以如果是相同值则不应该出栈。

</td></tr>
<tr><td>Arrays</td><td>
连续子数组和的计算

存所有前缀数组和<code>sum(0,i)</code>. 任意一个连续子数组的和<code>sum(i,j)=sum(0,j)-sum(0,i)</code>。把前缀数组和存到set里方便查找（查的时候减一下），如是否是一个数K的倍数。

注意初始化存一个0代表空集的和，便于查找<code>sum(0,i)</code>本身

Two Pointer
相向型

模板如下
<pre>left, right = 0,n-1
while (left < right):
    if (num[left], num[right 满足某条件):
        right--, 因为[left+1,right-1]和right的组合也满足该条件，可略过
    else:
        left++, 如上同理</pre>
NSum两种解法

排序+相向指针。适用于N>3，除去排序是O(N)时间，O(1)空间
HashSet。O(N)时间和空间
2N-Sum:可以分解成两个NSum以HashSet的形式解决

Partition型

不稳定算法，无法保证值的相对顺序
如果分成两个区，则两个指针。low指向低值区结尾（小于low全是低值，low自己未知），high指向高值区结尾。low若碰到高值则swap(low,high)，high--，反之low++
如果分成三个区，则在二区基础增加一个指针cur指向中值区结尾，移动cur并类似和low/high交换。

扫描线

用于解决多个区间重叠，求重叠状态/最大重叠个数等的情况
排序区间的起点/终点。有一条垂直于坐标轴的线从左到右扫描每个起点/终点。起点则重叠区间数自增，终点自减。

</td></tr>
<tr><td>Complexity</td><td>
Master Theorem
<code>T(n) = a*T(n/b) + n^d</code>, 一共有<code>log_b n</code>层

<code>a=b^d: T(n) = n^d * logn</code>, 因为每层加起来都是<code>n^d</code>. Merge Sort: <code>T(n) = 2T(n/2)+n = O(n*logn)</code>
<code>a<b^d : T(n) = n^d</code>,因为为每层等比数列逐渐减小,取第一层。 Quick Select: <code>T(n) = T(n/2)+n = O(n)</code>
<code>a>b^d : T(n) = n^logba</code>,因为为每层等比数列逐渐增加,取第最后一层。理解：最后一层计算量是<code>a^logbn * n'^d</code>, n'=1因为是递归末尾。所以是<code>a^logbn=n^logba</code>

</td></tr>
<tr><td>Advanced Algorithms</td><td>
快速选择(quick select)

用于Top K/Find Median问题，跟快排一样选pivot。不同的是只用处理一边，所以时间复杂度是O(n)

</td></tr>
</table>

OOD

几个原则	• **Encapsulation**: Getters and setters, don't expose data directly

	○ Private attributes: can't be accessed by outside/subclasses. Prefix with _(single underscore)
	○ Protected attributes: can't be accessed by outside, but subclass can. Prefix with __(double underscore)
	`class Class:`
	`  def set_attribute(self, attribute):`
	`  self.__attribute = attribute`
	`class.__attribute = 'value' # won't change the value`
	`class.set_attribute('value') # will work`
	● **Abstraction:** define interface with methods, without actual implementation. Example is shape + draw method
	○ Python DOES NOT have an Interface. So interface is implemented by defining an empty class with abstract methods
	`class Interface:`
	`  @abstractedmethod # important`
	`  def method1(self):`
	`  pass`
	`  @abstractedmethod`
	`  def method2(self):`
	`  pass`
	
	`class InvalidImplementation:`
	`  @abstractedmethod`
	`  def method1(self):`
	`  print('method 1)`
	
	`instance = InvalidImplementation() #系统报错 ,因为没有实现带有@abstractmethod的method2`
	
	● **Inheritance:** subclass inherits parent class's method/data
	○ `class Employee(Person): # Employee inherits Person. Can Multi-inherit`
	`  def __init__(self, name, salary):`
	`  self.salary = salary`
	`  Person.__init__(name) #call parent class's constructor`
	
	● **Polymorphism:**
	○ For Class: multiple classes have different implementation with same method name(doesn't have to be subclass)
	○ For Function: same function can take multiple types

Class/Static variables	● Any variable defined under class is class variable; any variable defined within method is instance variable
	● If you change class variable for an instance, it won't affect other instances; but if you change it for the class, it will affect everything
	`class Student:`
	`  department = 'EE'`
	
	`  def __init__(self):`
	`  return`
	
	`s1 = Student()`
	`s2 = Student()`
	`s1.department = 'CS' # won't affect s2`
	`Student.department = 'CS' # will affect s2`

Useful functions	● `issubslass: input subclass and parent class` ○ `issubclass(childClass, parentClass)` ○ `issubclass(type(childInstance), type(parentInstance))` ● `type: get the type(class) of an instance.`