基本类型操作

基本类型转换：

```
int(x)  # 转为整数
float(x) # 转为浮点数
complex(r, i) #创建复数

str(x) #转为字符串
repr(x) #将x装换为表达式字符串

list(s) #转成列表
set(s) #转成元组
dict(d) #转成字典，d是一个（key,value）的元组序列

chr(x) #整数转字符
ord(x) #字符转整数
hex(x) # 转十六进制
oct(x) # 转八进制

eval("2 + 2") #函数用来执行一个字符串表达式，并返回表达式的值。
```

python的输入输出

输入字符串：

```
name = input()

# 可用 int() 将输入字符串转成数字
n = int(input())

# 输入多个数字
m, n = map(int, input().split(' '))

# 循环输入
while(True):
    try:
        a = input()
        print(a)
    except:
        break

# 输出:结尾默认换行，中间默认空格（可修改）
print(name, "Mary", sep=",", end="。")
```

格式化输出：

浮点数输出

%f ——保留小数点后面六位有效数字

%.3f，保留3位小数位

%e ——保留小数点后面六位有效数字，指数形式输出

%.3e，保留3位小数位，使用科学计数法

%g ——在保证六位有效数字的前提下，使用小数方式，否则使用科学计数法

%.3g，保留3位有效数字，使用小数或科学计数法

```
print("%d" % 20)  #整数
print("%o" % 20)  #八进制
print("%x" % 20)  #十六进制

print('%.1f' % 1.11)

round(1.1135,3)  # 取3位小数，四舍五入
```

字符串输出

%s

%10s——右对齐，占位符10位

%-10s——左对齐，占位符10位

%.2s——截取2位字符串

%10.2s——10位占位符，截取两位字符串

```
>>> print('%20s' % 'hello world')  # 右对齐，取20位，不够则补位
>>> print('%-20s' % 'hello world')  # 左对齐，取20位，不够则补位
>>> print('%.2s' % 'hello world')  # 取2位
```

format用法:

位置匹配：

（1）不带编号，即“{}”

（2）带数字编号，可调换顺序，即“{1}”、“{2}”

（3）带关键字，即“{a}”、“{tom}”

```
print('{0} {1} {0}'.format('hello','world'))  # 打乱顺序
print('{a} {tom} {a}'.format(tom='hello',a='world'))  # 带关键字
```

格式化输出：

'b' - 二进制。将数字以2为基数进行输出。

'c' - 字符。在打印之前将整数转换成对应的Unicode字符串。

'd' - 十进制整数。将数字以10为基数进行输出。

'o' - 八进制。将数字以8为基数进行输出。

'x' - 十六进制。将数字以16为基数进行输出，9以上的位数用小写字母。

'e' - 幂符号。用科学计数法打印数字。用'e'表示幂。

'g' - 一般格式。将数值以fixed-point格式输出。当数值特别大的时候，用幂形式打印。

'n' - 数字。当值为整数时和'd'相同，值为浮点数时和'g'相同。不同的是它会根据区域设置插入数字分隔符。

'%' - 百分数。将数值乘以100然后以fixed-point('f')格式打印，值后面会有一个百分号。

```
print('{:.3f}'.format(20))
```

左中右对齐及位数补全:

（1）< （默认）左对齐、> 右对齐、^ 中间对齐、= （只用于数字）在小数点后进行补齐

（2）取位数“{:4s}”、"{:.2f}"等

```
print('{} and {}'.format('hello','world'))  # 默认左对齐
```

正负符号显示

```
'{:+f}; {:+f}'.format(3.14, -3.14)  # 总是显示符号

'Correct answers: {:.2%}'.format(points/total)

# 显示时间
import datetime
'{:%Y-%m-%d %H:%M:%S}'.format(datetime.datetime.now())
```

字符串

```
# 1、去空格及特殊符号
# strip() 方法用于移除字符串头尾指定的字符（默认为空格
s.strip().lstrip().rstrip(',')

# 2、复制字符串
s1 = "ab"
s2 = s1

# 连接字符串
s1 += s2

# 查找字符
pos = s.index(c)

# 比较字符串
s1==s2

S.lower() #小写
S.upper() #大写
S.swapcase() #大小写互换
S.capitalize() #首字母大写

# 翻转字符串
s = s[::-1]

# 查找字符串
s.find(sub_s)  #返回索引

# 分割字符串
s.split(',')

# 删空格
s.replace(' ', '')

# 连接字符串列表
','.join(lst)

# 统计空格出现的数量
s.count(' ')

# 内置字符串
```

```
string.digits #0123456789

string.ascii_letters #abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

string.ascii_lowercase #abcdefghijklmnopqrstuvwxyz

string.ascii_uppercase #ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

列表

列表是一个数据的集合，可以放任何数据类型，可方便的增删改查。

```
lst = []

lst.append(1)  #给列表后面追加新元素

lst.clear()    #清空列表

lst.copy()     #浅复制表（只复制一层）

lst.count(val) #统计某个元素在列表中出现的次数

lst.extend(lst2) #表尾追加一个序列

lst.index(val)  #从列表中找出某个值第一个匹配项的索引位置

lst.insert(index, val) #插入列表中index

lst.pop() # 移除列表中的一个元素（默认最后一个），并返回该元素的值
lst.pop(index)

lst.remove(value) #移除列表中某个值的第一个匹配项，从左找到第一个指定元素

lst.reverse() #翻转列表中的元素
lst.sort(key=None) #排序原列表

# 删除列表元素
del lst[index]
lst.pop(index)
```

列表切片：

[:] 提取从开头（默认位置0）到结尾（默认位置-1）的整个字符串
[start:] 从start 提取到结尾
[: end] 从开头提取到end - 1
[start : end] 从start 提取到end - 1
[start : end : step] 从start 提取到end - 1，每step 个字符提取一个
• 左侧第一个字符的位置为0，右侧最后一个字符的位置-1

直接声明的list和tuple无法通过dict()转换成dict类型；直接声明的dict可以通过tuple()和list()分别转换成tuple和list类型（结果只包含了keys），可是此时却能通过dict()反转回原来的dict类型。

元组

Python 的元组与列表类似，不同之处在于元组的元素不能修改。

```
tup1 = ()

#只有一个元素，要在元素后添加逗号
tup1 = (50, )


# 可以使用下标索引来访问元组中的值
tup1[0]

# 修改元组:对元组进行连接组合
tup3 = tup1 + tup2

# 删除元组：del语句来删除整个元组
del tup1
```

集合

集合（set）是一个**无序**的不重复元素序列。

```
set1 = set()
set2 = {1,2,3}

# 添加元素
s.add(x) #若元素已存在，则不进行任何操作

s.update(x) #x可以有多个，用逗号分开

# 移除元素
s.remove(x) # 将元素 x 从集合 s 中移除

s.discard(x) # 若不存在不会发生错误

# 集合元素个数
len(s)

s.clear()

x in s

## 集合的操作

# 两个集合是否包含，子集操作：
s1.issubset(s2) # s1<=s2;
s2.issuperset(s1) # s2 >= s1

# 集合的并集 s1|s2
s1.union(s2)

# 集合的交集 s1&s2
s1.intersection(s2)

s1.difference(s2) # 包含s1中有，但是s2没有的元素的集合。s1-s2

s1.symmetric_difference(s2) #包含s1和s2中不相同的元素的集合。(s1-s2)|(s2-s1) == s1^s2 集合的异或
```

frozenset() 返回一个冻结的集合，冻结后集合不能再添加或删除任何元素。

字典

字典中每个成员是以“键：值”对的形式存放具有映射关系的数据。

- 字典是无序的，它不能通过偏移来存取，只能通过键来存取；
- 键必须是唯一；
- 键必须是不可变的数据类型，比如，数字，字符串，元组等，列表等可变对象不能作为键；
- 值可以是任意类型的对象；

```python
dct = {}
scores_dict = {'语文': 105, '数学': 140, '英语': 120}

# 访问字典里的值
scores_dict['语文'] # 通过键索引,不存会报错

scores_dict.get('历史') # get() 方法访问不存在的 key，该方法会简单地返回 None

# 添加键值对
scores_dict['物理'] = 97

scores_dict.update({'语文': 120, '数学': 110}) #使用一个字典所包含的 key-value 对来更新已有的字典

# 删除键值对
del scores_dict['数学']
scores_dict.pop('英语') # # 删除'英语'的键和值
scores_dict.popitem() # 弹出字典中最后一个key-value对

# 修改值
scores_dict['数学'] = 120

#  判断键值对是否存在
'语文' in scores_dict # key in dct
'历史' not in scores_dict # not in

# 清空
scores_dict.clear()

# 以列表返回可遍历的(键, 值) 元组数组
scores_dict.items()

# 以列表返回一个字典所有的键
scores_dict.keys()

# 以列表返回字典中的所有值
scores_dict.values()

scores_dict.setdefault('语文', 100) # key 在字典中不存在时，设置'语文'默认值为100

scores_dict = dict.fromkeys(['语文', '数学']) #方法使用给定的多个key创建字典， key对应的value默认
scores_dict = dict.fromkeys(('语文', '数学'), 100) #指定默认值100
```

```
len(scores_dict) #键的总数。
```

字典的for循环遍历:

```
# 遍历key的值
for key in scores_dict:
    print(key)

# 遍历value的值
for value in scores_dict.values():
    print(value)

# 遍历字典键值对
for key in scores_dict:
    print(key + ":" + str(scores_dict[key]))

for key, value in scores_dict.items():
    print(key + ':' + str(value))
```

字典的迭代构建：

```python
# 统计出现次数
nums = [1,2,2,3]
dct = dict.fromkeys(nums, 0)
for x in nums:
    dct[x] += 1

nums = [1,2,2,3]
dct = {}
for x in nums:
    dct[x] = dct.get(x, 0) + 1


from collections import defaultdict
dct = defaultdict(list)


# 迭代构建时要考虑key不存在时，value的默认设置，特别当value为列表或字典时
dct.setdefault(key, []).append(value) # 值为列表

dct.setdefault(key, {})[value] = weight # 值为字典
```

堆栈

可将列表当做堆栈使用
列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。

- 用 append() 方法可以把一个元素添加到堆栈顶。
- 用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。

队列

Queue的种类：

- Queue.Queue() 先进先出(FIFO)
- Queue.LifoQueue() 后进先出(LIFO)
- Queue.PriorityQueue() 优先队列

```
from queue import Queue
q = Queue()

# 入队
q.put(i)

# 出队(移除返回)
q.get()

# 队列大小
q.qsize()

# 判断队空
q.empty()
```

堆

heapq 模块，默认是最小堆结构，可以直接利用其中的函数操作来实现最小优先队列的功能。

```
arr = [1, 2, 3]

# 列表转为最小堆
heapq.heapify(arr)

# 插入元素
heapq.heappush(arr, 5)

# 取堆顶（弹出最小元素）
item = heapq.heappop(arr)
```

最大堆：**通过把元素取反再放入堆，出堆时再取反**，把问题转换为最小堆问题也能间接实现最大堆。

双端队列

deque 是Python标准库 collections 中的一个类，实现了两端都可以操作的队列，相当于双端队列，与Python的基本数据类型列表很相似。

```python
from collections import deque

dq = deque()

# 左添加(头部)
dq.appendleft("A")

# 右添加（尾部）
dq.append('a')

# 多个元素入队
dq.extendleft(['c', 'd'])
dq.extend(['D', 'E'])

# 队头弹出元素
dq.popleft()

# 队尾弹出元素
dq.pop()

# 删除指定元素
dq.remove(x)

#  旋转
# 若num>=1，表示从右向左的num个数，与其左边的所有数顺时针旋转
# 若num<=-1，表示从左向右的-num个数，与其右边的所有数逆时针旋转

dq.rotate(3) # 从右向左的num个数顺时针旋转
```

链表

```
# 双向链表的结点定义
class Node:
    def __init__(self, val = 0, prev = None, next = None):
        self.val = val
        self.prev = prev
        self.next = next

class DLinkedList:
    def __init__(self):
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head
    def move_to_tail(self, node):
        '''
        添加到表尾
        '''
        self.tail.prev.next = node
        node.prev = self.tail.prev
        self.tail.prev = node
        node.next = self.tail

    def create_from_list(self, arr):
        for x in arr:
            node = Node(x)
            self.move_to_tail(node)

    def print_DLL(self):
        print("顺序打印：")
        p = self.head.next
        while(p != self.tail):
            print(p.val, end=' ')
            p = p.next
        print("\n逆序打印：")
        p = self.tail.prev
        while(p != self.head):
            print(p.val, end = ' ')
            p = p.prev
```

二叉树

二叉树结点定义：

```
class TreeNode:
    def __init__(self, val = 0, left = None, right = None):
        self.val = val
        self.left = left
        self.right = right
```

图

建图

```python
class Graph:
    def __init__(self):
        self.graph = {}

    def create_from_directed(self, data):

        for src, dst, weight in data:
            # self.graph[src].update({dst:weight})
            self.graph.setdefault(src,{})[dst] = weight

    def create_from_undirected(self, data):
        for p1, p2, weight in data:
            self.graph.setdefault(p1,{})[p2] = weight
            self.graph.setdefault(p2,{})[p1] = weight

if __name__ == "__main__":
    data = [['A', 'B', 2],
            ['A', 'C', 4],
            ['B', 'D', 3],
            ['C', 'F', 4],
            ['D', 'E', 1],
            ['E', 'F', 3]]
    g = Graph()
    # g.create_from_directed(data)  # 有向
    g.create_from_undirected(data)  # 无向
    print(g.graph)
```

```
{'A': {'B': 2, 'C': 4}, 'B': {'A': 2, 'D': 3}, 'C': {'A': 4, 'F': 4}, 'D': {'B': 3, 'E':
```

Python 闭包

函数在某个作用域内调用，如何处理自由变量？

Python设计：在函数定义体中赋值的变量是局部变量。

```
b = 6
def f(a):
    print(a)
    print(b)
    b = 9
# 出错
f(3)
```

若要将b当作全局变量，则需要使用global声明

```
def f(a):
    global b
    print(a)
    print(b)
    b = 9
```

类的实现不存在自由变量问题，因为用的是类属性。如果是函数嵌套：

```
def make_avg():
    lst = []

    def avg(new_val):
        lst.append(new_val)
        return sum(lst)/len(lst)
    return avg

a = make_avg()
a(10) #10.0
a(11) #10.5
a(12) #11.0
```

lst定义在make_avg函数的局部作用域中，make_avg()已经返回了，局部作用按理应该消失，执行a(10)会出错（lst找不到定义）

然而，Python通过闭包实现，保留定义时存在的自由变量的绑定，这样虽然定义作用域不可用，绑定依然可以使用

nolocal

```
def make_avg():
    count = 0
    total = 0

    def avg(new_val):
        nonlocal count, total #注意
        count += 1
        total += new_val
    return total/count

a = make_avg()
```

局部变量在赋值前进行了引用，count，total就变成了avg的局部变量，要将他们的作用域扩展到make_avg中，需要nonlocal关键字声明；