150道练习题，涵盖基础内容的方方面面

1.数据类型篇

本篇的练习题旨在考察你对基本数据类型的理解熟悉程度，适合刚接触python的初学者用来巩固对基础知识的理解

1.1 基本数据类型

1.1.1 逻辑推理练习（类型转换）

不运行程序，说出下面程序的执行结果

```
1. 4.0 == 4
2. "4.0" == 4
3. bool("1")
4. bool("0")
5. str(32)
6. int(6.26)
7. float(32)
8. float("3.21")
9. int("434")
10. int("3.42")
11. bool(-1)
12. bool("")
13. bool(0)
14. "wrqq" > "acd"
15. "ttt" == "ttt "
16. "sd"*3
17. "wer" + "2322"
```

答案如下

```
1. True
2. False
3. True
4. True
5. '32'
6. 6
7. 32.0
8. 3.21
9. 434
10. 会报错
11. True
12. False
13. False
```

```
14. True
15. False
16. "sdsdsd"
17. 'wer2322'
```

关于这些答案，要做到知其然且知其所以然，编程需要精准的知道每一个细节，下面对其中一些可能让你感到困惑的知识点进行讲解

1.1.1.1 bool函数转换规则

bool函数进行转换时，其结果取决于传入参数与True和False的等价关系，只需记住一点即可

0，空字符串，None在条件判断语句中等价于False，其他数值都等价于True

bool函数在做数据类型转换时遵循该原则

1.1.1.2 int("3.42") 为什么会报错

字符串"3.42"可以转成float类型数据3.42，3.42可以转成int类型数据3，但是字符串"3.42"却不可以直接使用int函数转成3，讲实话，我也觉得这个函数有些不灵活，或许是语言的发明者有自己的考虑吧，咱们对这种问题，不必深究，先做到知道它是什么，将来再去研究为什么

1.1.1.3 字符串大小比较规则

两个字符串在比较大小时，比的不是长度，而是内容

字符串左对齐后，逐个字符依次比较，直到可以分出胜负

1.1.1.4 "sd"*3

"sd"*3 的意思是sd重复3次，生成一个新的字符串

1.1.2 数据类型考察

请说出下面表达式结果的类型

```
1. "True"
2. "Flase"
3. 4 >= 5
4. 5
5. 5.0
6. True
```

非常简单的送分题

```
1. str
2. str
3. bool
4. int
5. float
6. bool
```

唯一需要解释的是4 >= 5,4比5小，怎么可能大于等于5呢，这是错误的，既然是错的，那么就等于False,False的类型是bool

1.1.3 交互式解释器练习

请在交互式解释器里回答下面的题目

```
1. 3的5次方
2. 7对2求模
3. 9除5，要求有小数部分
4. 9除5，要求没有小数部分
5. 用程序计算根号16，也就是16的2分之一次方
```

答案如下

```
1. 3**5
2. 7%2
3. 9/5
4. 9//5
5. import math
   math.sqrt(16)
```

知识点讲解

1. 幂运算用两个， *2的2次方表示为2*2*
2. 求模运算用%， 其实就是求余数，不知道余数的打电话给小学老师
3. 除法中，希望结果有小数部分时用/， 希望只保留整数部分时用 // ,没啥可解释的，请记住他们的区别，懒得记，就别学编程，编程不适合懒惰的人
4. 开根号，要用到math模块的sqrt方法，这个题目需要你自己去百度或是谷歌，第一次明确的建议你，一定要好好利用搜索引擎，不会用搜索引擎的程序员，永远是菜鸟

1.2 字符串练习题

1.2.1 字符串内置方法练习

在交互式解释器中完成下列题目

1. 将字符串 "abcd" 转成大写
2. 计算字符串 "cd" 在 字符串 "abcd"中出现的位置
3. 字符串 "a,b,c,d" ，请用逗号分割字符串，分割后的结果是什么类型的?
4. "{name}喜欢{fruit}".format(name="李雷") 执行会出错，请修改代码让其正确执行
5. string = "Python is good", 请将字符串里的Python替换成 python,并输出替换后的结果
6. 有一个字符串 string = "python修炼第一期.html"，请写程序从这个字符串里获得.html前面的部分，要用尽可能多的方式来做这个事情
7. 如何获取字符串的长度?
8. "this is a book",请将字符串里的book替换成apple
9. "this is a book", 请用程序判断该字符串是否以this开头
10. "this is a book", 请用程序判断该字符串是否以apple结尾
11. "This IS a book"， 请将字符串里的大写字符转成小写字符
12. "This IS a book"， 请将字符串里的小写字符，转成大写字符
13. "this is a book\n"， 字符串的末尾有一个回车符，请将其删除

在看答案之前，我要非常明确的告诉你，答案所涉及的每一个字符串方法，都是需要你记忆下来的，就像九九乘法表那样熟记于心，这不是要求，而是必须，否则，你凭什么说你会一门编程语言呢？ 聪明从来不自己骗自己!

答案如下

```
1. "abcd".upper()
2. "abcd".find('cd')
3. "a,b,c,d".split(',')
4. "{name}喜欢{fruit}".format(name="李雷", fruit='苹果')
5. string.replace('Python', 'python')
6. string[0:string.find('.html')] 或者string[0:-5]
7. 使用len函数
8. "this is a book".replace('book', 'apple')
9. "this is a book".startswith('this')
10. "this is a book".endswith('apple')
11. "This IS a book".lower()
12. "This IS a book".upper()
13. "this is a book\n".strip()
```

这里只对其中2个题目讲解

第4小题的程序直接运行会报错，因为字符串里面有两个需要替换的位置，而format方法里只传入了一个参数，显然是不够

第13小题，strip() 方法用于移除字符串头尾指定的字符（默认为空格或换行符）或字符序列， \n 就是换行符，这里又涉及到转义字符这个概念，本篇不做详细讲解，求知欲强的同学可以自己百度一下

1.2.2 逻辑推理练习（字符串）

不用代码，口述回答下面代码的执行结果
string = "Python is good"

1. string[1:20]
2. string[20]
3. string[3:-4]
4. string[-10:-3]
5. string.lower()
6. string.replace("o", "0")
7. string.startswith('python')
8. string.split()
9. len(string)
10. string[30]
11. string.replace(" ", '')

答案如下

```
1. 'ython is good'
2. 报错
3. 'hon is '
4. 'on is g'
5. 'python is good'
6. 'Pyth0n is g00d'
7. False
8. ['Python', 'is', 'good']
9. 14
10. 报错
11. 'Pythonisgood'
```

第2题和第10题都报错，是因为超出了索引范围，字符串长度为14，你去20和30的位置取值，当然会报错

关于切片操作，只需要知道从哪里开始到哪里结束就一定能推导出答案，以string[3:-4]为例，3是开始的位置，-4是结束的位置，但这个范围是左闭右开的，从3开始没错，但不会到-4，而是到-5，更前面的一个位置，python支持负数索引，或者说是反向索引，从右向左从-1开始逐渐减小。

第一题中，做切片的时候是从1开始，到20结束，即便是右开，直到19，也仍然超出了索引范围，为什么不报错呢，这就是语言设计者自己的想法了，切片时，不论是开始位置还是结束位置，超出索引范围都不会报错，我猜，大概是由于切片是一个范围操作，这个范围内有值就切出来，没值返回空字符串就好了。

1.3 列表与元组练习题

1.3.1 列表基础考察

已知一个列表
lst = [1,2,3,4,5]

1. 求列表的长度
2. 判断6 是否在列表中
3. lst + [6, 7, 8] 的结果是什么?
4. lst*2 的结果是什么
5. 列表里元素的最大值是多少
6. 列表里元素的最小值是多少
7. 列表里所有元素的和是多少
8. 在索引1的后面新增一个的元素10
9. 在列表的末尾新增一个元素20

答案如下

```
1. len(lst)
2. 6 in lst
3. [1,2,3,4,5,6,7,8]
4. [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
5. max(lst)
6. min(lst)
7. sum(lst)
8. lst.insert(1, 10)
9. lst.append(20)
```

以上都是对列表基础操作，所用到的每一个函数，列表的每一个方法，都是需要你熟记于心的

1.3.2 修改列表

lst = [1, [4, 6], True]
请将列表里所有数字修改成原来的两倍

答案如下

```
lst[0] = 2
lst[1][0] = 4
lst[1][1] = 12
```

你以为存在一个函数，其功能便是将列表里所有的数据都变成原来的两倍，这样才显得变成语言是一个非常神奇的东西，但是很遗憾的告诉你，那些神奇的东西都是程序员自己实现的。

想要修改列表里的数据，必须通过索引对其重新赋值，上面的方法很low，你也可以写一个函数来实现这个功能，我们假设要处理的列表里只int,float,bool,和list数据，不管嵌套基层list，这个函数都应该能正确处理，下面是一段示例代码

```
def double_list(lst):
    for index, item in enumerate(lst):
        if isinstance(item, bool):
            continue
        if isinstance(item, (int, float)):
            lst[index] *= 2
        if isinstance(item, list):
            double_list(item)


if __name__ == '__main__':
    lst = [1, [4, 6], True]
    double_list(lst)
    print(lst)
```

1.3.3 元组概念考察

写出下面代码的执行结果和最终结果的类型

1. (1, 2)*2
2. (1,)*2
3. (1)*2

答案如下

```
1. (1, 2, 1, 2)
2. (1, 1)
3. 2
```

第一题应该没有异议，关键是第2题和第3题，元组里只有一个数据时，必须有逗号，如果没有逗号，就变成了第3题的形式，第3题本质上是1*2,那对小括号就如同我们小学学过的小括号一样，只是为了体现运算优先级而已。

当元组只有一个数据时，如果不省略了逗号，那么小括号的作用就不再是表示元组，而是表示运算优先级

1.3.4 合并列表

lst = [1,2,3]
lst2 = [4,5,6]
不使用 + 号运算符，将lst2合并到lst的末尾，并思考，这个过程中，是否产生了新的列表

答案

```
lst.extend(lst2)
```

这个过程中不会产生新的列表，最直观的检验方式就是print(id(lst)),合并前后，lst的内存地址都没有发生变化，只是列表里的内容发生了变化

1.3.5 合并字符串

str1 = "1,2,3"
str2 = "4,5,6"
请将str2合并到str1的末尾，并思考，这个过程中，是否产生了新的字符串

答案

```
str1 += str2
```

这个过程中，产生的新的字符串，字符串是不可变对象，从字面上理解，似乎str1的内容发生变化了，但本质上是产生了新的字符串并赋值给str1， print(str1)， 合并前后的内存地址是不一样的

1.3.6 统计练习

列表lst 内容如下

```
lst = [2, 5, 6, 7, 8, 9, 2, 9, 9]
```

请写程序完成下列题目

1. 找出列表里的最大值
2. 找出列表里的最小值
3. 找出列表里最大值的个数
4. 计算列表里所有元素的和
5. 计算列表里元素的平均值
6. 计算列表的长度
7. 找出元素6在列表中的索引

答案

```
1. max(lst)
2. min(lst)
3. lst.count(max(lst))
4. sum(lst)
5. sum(lst)/float(len(lst))
6. len(lst)
7. lst.index(6)
```

这道题考察的是你对内置函数的理解和运用

下面的题目不允许写代码，仅凭思考来回答

1. lst[2:4] 的值是什么
2. lst[1: −3]的值是什么
3. lst[−5]的值是什么
4. lst[:−4] 的值是什么
5. lst[−4:] 的值是什么

这个题目主要考察你对列表切片操作的理解

```
1. [6, 7]
2. [5, 6, 7, 8, 9]
3. 8
4. [2, 5, 6, 7, 8]
5. [9, 2, 9, 9]
```

列表的切片操作，最关键的一点在于左闭右开，结束位置的数据不会列入结果中

1.3.7 列表操作练习

列表lst 内容如下

```
lst = [2, 5, 6, 7, 8, 9, 2, 9, 9]
```

请写程序完成下列操作

1. 在列表的末尾增加元素15
2. 在列表的中间位置插入元素20
3. 将列表[2, 5, 6]合并到lst中
4. 移除列表中索引为3的元素
5. 翻转列表里的所有元素
6. 对列表里的元素进行排序，从小到大一次，从大到小一次

答案

```
1. lst.append(15)
2. lst.insert(len(lst)//2, 20)
3. lst.extend([2, 5, 6])
4. lst.remove(lst[3])
5. lst = lst[::-1]
6. lst.sort()   lst.sort(reverse=True)
```

1.3.8 复杂列表练习

列表lst 内容如下

```
lst = [1, 4, 5, [1, 3, 5, 6, [8, 9, 10, 12]]]
```

不写任何代码，仅凭思考来回答下列问题

1. 列表lst的长度是多少
2. 列表lst中有几个元素
3. lst[1] 的数据类型是什么
4. lst[3]的数据类型是什么
5. lst[3][4] 的值是什么
6. 如果才能访问到 9 这个值
7. 执行lst[3][4].append([5, 6])后，列表lst的内容是什么，手写出来
8. lst[-1][-1][-2]的值是什么

9. lst[-2]的值是什么
10. len(lst[-1]) 的值是什么
11. len(lst[-1][-1])的值是什么
12. lst[-1][1:3] 的值是什么
13. lst[-1][-1][1:-2]的值是什么

第1题和第2题其实是一个意思，原本统计列表里数据个数不是什么难事，可一旦出现了嵌套列表的情况，有人就分不清了，列表里的数据是以逗号分隔的，lst[3] 是一个列表，其余都是int类型数据，因此lst的长度是4

第3题，lst[1] = 4,是int类型数据
第4题，lst[3] 的数据类型是列表
第5题，lst[3]的值是[1, 3, 5, 6, [8, 9, 10, 12]]，仍然是一个列表，其索引为4的数据是[8, 9, 10, 12]，是列表
第6题，lst[3][4][1]
第7题，[1, 4, 5, [1, 3, 5, 6, [8, 9, 10, 12, [5, 6]]]],参考5，6两个题目的解答
第8题，lst[-1]的值是[1, 3, 5, 6, [8, 9, 10, 12]]， 再次取索引为-1的数据为[8, 9, 10, 12]，取索引为-2的数据为10
第9题，5
第10题，5
第11题，4
第12题， [3, 5]， lst[-1]的值是[1, 3, 5, 6, [8, 9, 10, 12]]
第13题，[9]， lst[-1][-1]的值是[8, 9, 10, 12]，切片起始位置索引是1，值为9，结束位置是-2，值为10，由于左闭右开，最终结果是[9]

1.4 字典练习题

1.4.1 字典基本操作

字典内容如下

```
dic = {
    'python': 95,
    'java': 99,
    'c': 100
}
```

用程序解答下面的题目

1. 字典的长度是多少
2. 请修改'java' 这个key对应的value值为98

3. 删除 c 这个key
4. 增加一个key-value对，key值为 php, value是90
5. 获取所有的key值，存储在列表里
6. 获取所有的value值，存储在列表里
7. 判断 javascript 是否在字典中
8. 获得字典里所有value 的和
9. 获取字典里最大的value
10. 获取字典里最小的value
11. 字典 dic1 = {'php': 97}， 将dic1的数据更新到dic中

第1题，len(dic),结果为3
第2题，dic['java'] = 98,对字典里value的修改，必须通过key才可以
第3题，del dic['c']
第4题，dic['php'] = 90
第5题，lst = list(dic.keys())
第6题，lst = list(dic.values())
第7题，'javascript' in dic
第8题，sum(dic.values())
第9题，max(dic.values())
第10题，min(dic.values())
第11题，dic.update(dic1)

1.4.2 字典应用（买水果）

小明去超市购买水果，账单如下

```
苹果  32.8
香蕉  22
葡萄  15.5
```

请将上面的数据存储到字典里，可以根据水果名称查询购买这个水果的费用

很简单哦，用水果名称做key，金额做value，创建一个字典

```
info = {
    '苹果':32.8,
    '香蕉': 22,
    '葡萄': 15.5
}
```

1.4.3 字典应用（买水果2）

小明，小刚去超市里购买水果

小明购买了苹果，草莓，香蕉，一共花了89块钱，，小刚购买了葡萄，橘子，樱桃，一共花了87块钱

请从上面的描述中提取数据，存储到字典中，可以根据姓名获取这个人购买的水果种类和总费用。

以姓名做key，value仍然是字典

```
info = {
    '小明': {
        'fruits': ['苹果', '草莓', '香蕉'],
        'money': 89
    },
    '小刚': {
        'fruits': ['葡萄', '橘子', '樱桃'],
        'money': 87
    }
}
```

1.5 集合练习题

集合间的运算

```
lst1 = [1, 2, 3, 5, 6, 3, 2]
lst2 = [2, 5, 7, 9]
```

- 哪些整数既在lst1中，也在lst2中
- 哪些整数在lst1中，不在lst2中
- 两个列表一共有哪些整数

虽然题目一直在问两个列表，但用列表解答这3个题目效率很低，你应该用集合

```
lst1 = [1, 2, 3, 5, 6, 3, 2]
lst2 = [2, 5, 7, 9]

set1 = set(lst1)
set2 = set(lst2)

# 哪些整数既在lst1中，也在lst2中
print(set1.intersection(set2))
```

```
# 哪些整数在lst1中，不在lst2中
print(set1.difference(set2))

# 两个列表一共有哪些整数
print(set1.union(set2))
```

2. 基础语法篇

基础语法篇的练习题，不涉及复杂的逻辑推理，旨在检查你对基础语法的掌握情况

2.1 if 条件语句

2.1.1 单个条件分支

使用input函数接收用户的输入，如果用户输入的整数是偶数，则使用print函数输出"你输入的整数是:{value}, 它是偶数", {value}部分要替换成用户的输入。

完成这个练习题需要你掌握下面4个知识点

1. input函数的作用
2. 字符串转int
3. 取模运算
4. 字符串格式化

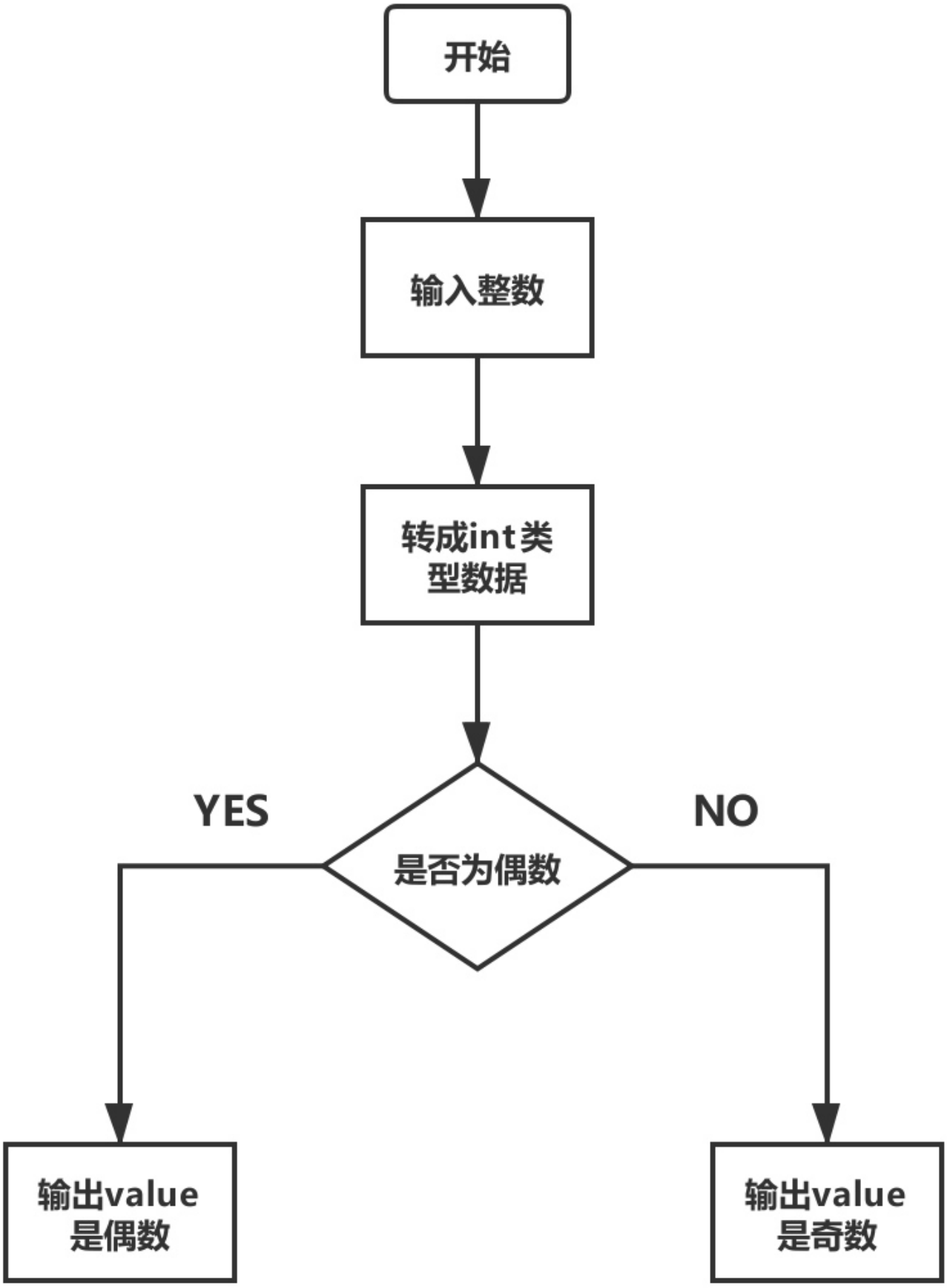
```
value = input("请输入一个整数:")
i_value = int(value)
if i_value % 2 == 0:
    print("你输入的整数是:{value}, 它是偶数".format(value=value))
```

2.1.2 if ... else ...

使用input函数接收用户的输入，如果用户输入的整数是偶数，则使用print函数输出"你输入的整数是:{value}, 它是偶数",如果是奇数，则使用print函数输出"你输入的整数是:{value}, 它是奇数"

```
value = input("请输入一个整数:")
i_value = int(value)
if i_value % 2 == 0:
    print("你输入的整数是:{value}, 它是偶数".format(value=value))
```

```
else:
    print("你输入的整数是:{value}, 它是奇数".format(value=value))
```

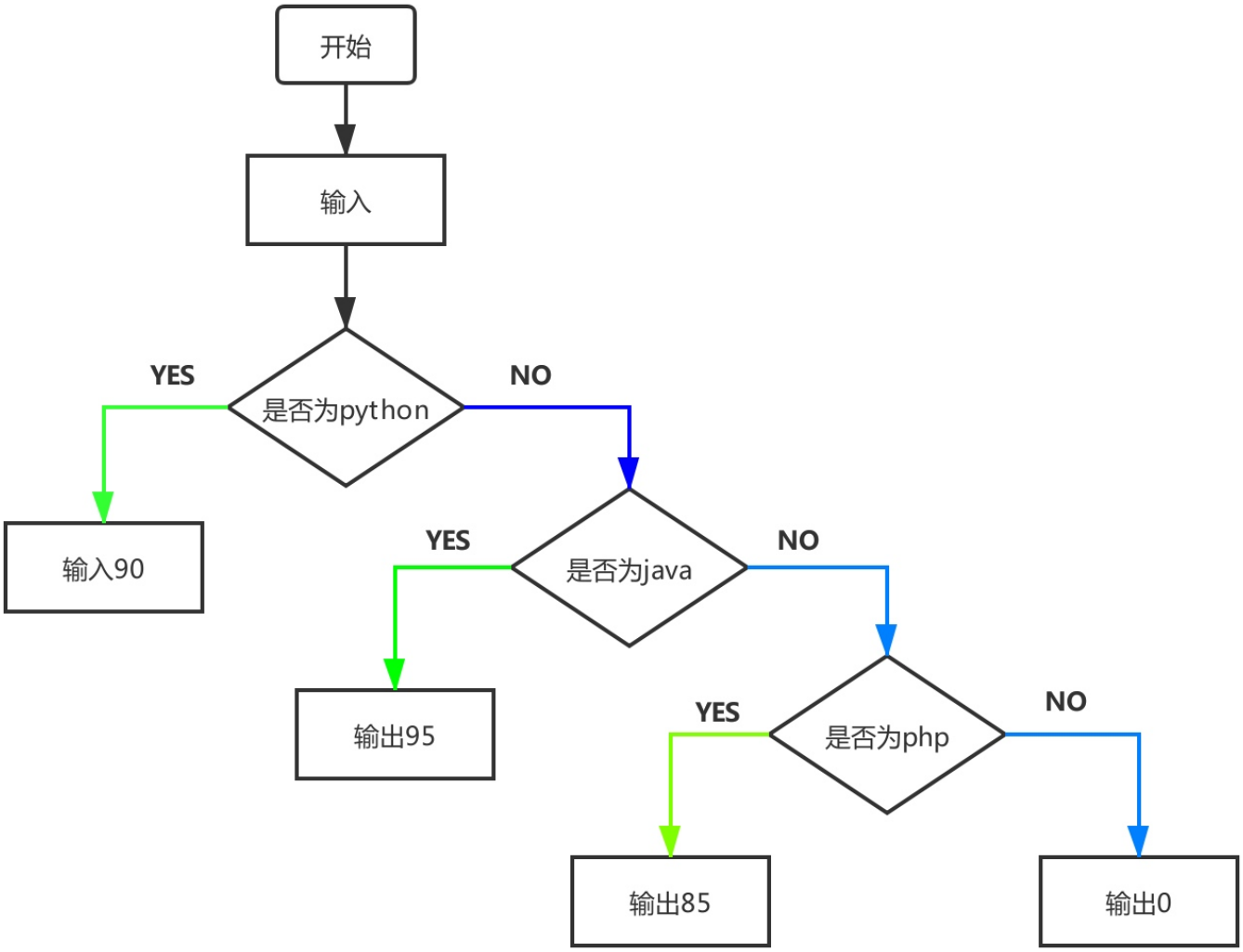


2.1.3 多条件分支

使用input函数接收用户的输入数据，如果用户输入python，则输出90， 如果用户输入java，输出95，如果用户输入php，输出85，其他输入，程序输出0

```
value = input("请输入一个整数:")

if value == 'python':
    print(90)
elif value == 'java':
    print(95)
elif value == 'php':
    print(85)
else:
    print(0)
```



2.1.4 复杂条件判断

使用input函数接收用户的输入，如果输入的数据不可以转换成int类型数据，则输出"无法使用

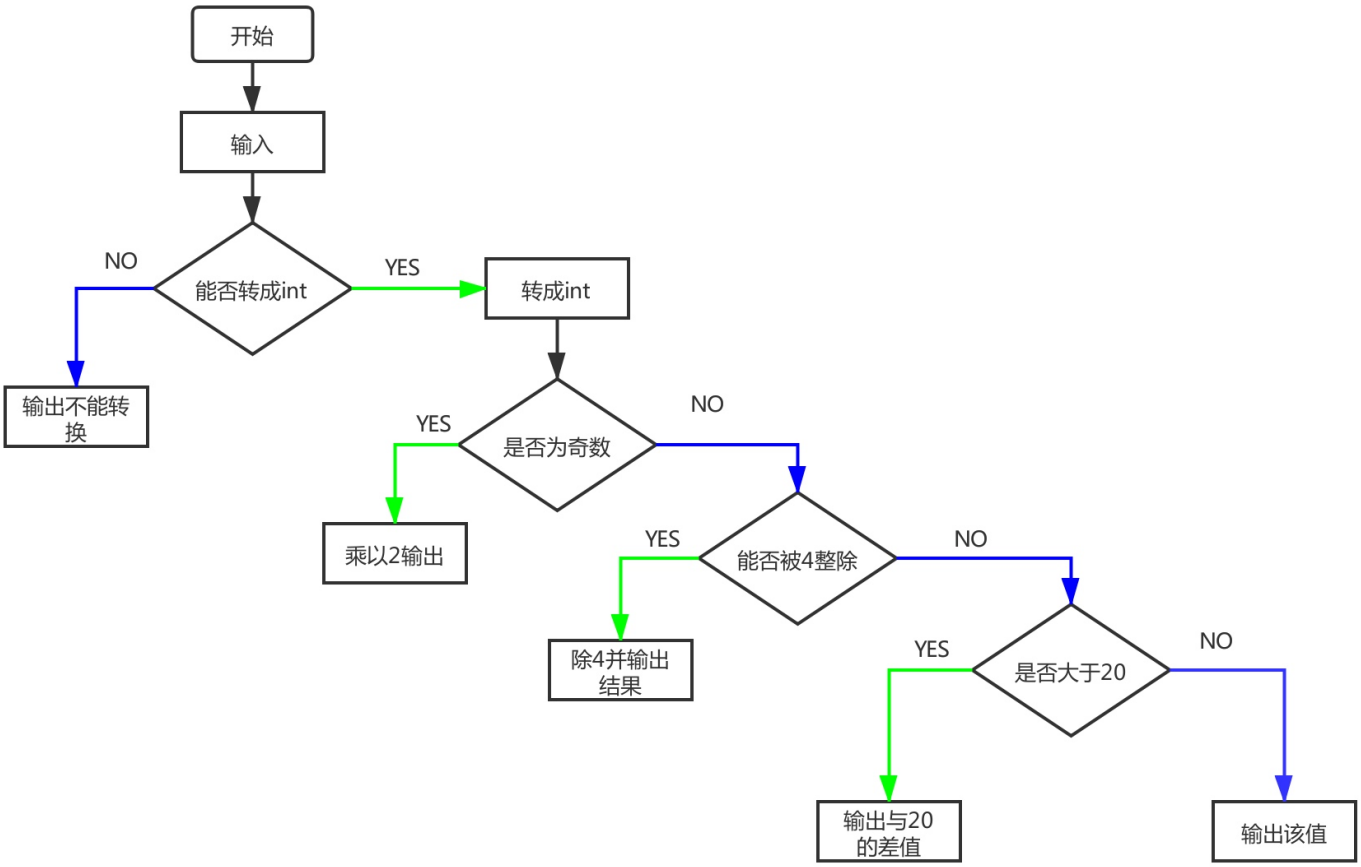
int函数转换"，如果可以，则将用户的输入转成int类型数据并继续判断。

如果输入数据是奇数，则将其乘以2并输出，如果是偶数，则判断是否能被4整除，如果可以则输出被4整除后的值，若不能被4整数，则判断是否大于20，如果大于20则输出与20的差值，如果小于等于20，则直接输出该值

程序代码如下

```
value = input("请输入一个整数:")
if not value.isdigit():
    print('无法使用int函数转换')
else:
    i_value = int(value)
    if i_value % 2 == 1:
        print(i_value*2)
    elif i_value % 4 == 0:
        print(i_value / 4)
    elif i_value > 20:
        print(i_value - 20)
    else:
        print(i_value)
```

画出程序流程图如下



2.2 for循环

2.2.1 range函数基本使用

```
range(3, 20, 4)
range(10, -3, -4)
range(10, 5)
range(2, 12)
```

不使用程序，说出上面4个range产生的整数序列

2.2.2 利用range函数遍历列表

```
lst = [1, 3, 5, 2, 7, 9]
for index in range(len(lst)):
    print(lst[index])
```

1. 参照上面的代码，从后向前遍历
2. 遍历输出列表里的所有偶数
3. 遍历列表，输出大于3的奇数

2.2.3 使用for循环遍历字典

遍历字典有两种方法
方法1

```
dic = {
    'python': 90,
    'java': 95
}

for key in dic:
    print(key, dic[key])
```

方法2

```
dic = {
    'python': 90,
    'java': 95
```

```
}

for key, value in dic.items():
    print(key, value)
```

2.2.4 continue练习

2.2.5 break练习

从列表 lst = [1, 3, 5, 2, 7, 9, 10] 中寻找1个偶数并输出，代码如下

```
lst = [1, 3, 5, 2, 7, 9, 10]
for item in lst:
    if item % 2 == 0:
        print(item)
        break
```

题目要求寻找一个偶数，当找到这个偶数后，循环就可以终止了，使用break可以终止本次循环，你可以去掉代码中的break，再次执行代码，观察代码的执行效果

2.2.6 寻找列表中的最大值,最小值

```
lst = [3, 6, 1, 8, 1, 9 , 2]

max_value = lst[0]
for item in lst:
    if item > max_value:
        max_value = item

print(max_value)
```

1. 参照上面的代码，写代码寻找列表的最小值
2. 写代码寻找列表里的最小偶数
3. 写代码寻找列表里的最大奇数

2.2.7 寻找组合

```
lst1 = [3, 6, 1, 8, 1, 9 , 2]
lst2 = [3, 1, 2, 6, 4, 8, 7]

for item1 in lst1:
```

```
    for item2 in lst2:
        if item1 + item2 == 10:
            print((item1, item2))
```

上面的代码利用嵌套循环，从两个列表里各取1个数，如果这两个数的和等于10，则以元组的方式输出这两个数

1. 参照上面的代码，寻找两个数的差的绝对值等于2的组合
2. 两个列表里各取出一个值，item1和item2， 请计算item1*item2的最大值

2.3 while循环

2.3.1 奇偶数判断

使用input函数接收用户输入的整数，如果是偶数，则使用print函数输出"你输入的是一个偶数",反之输出"你输入的是一个奇数"，用户可以输入多次，直到输入quit时程序退出

```
while True:
    input_str = input("请输入一个正整数，想退出程序请输入 quit:")
    if input_str == "quit":
        break
    number = int(input_str)

    if number % 2 == 0:
        print("你输入的是一个偶数")
    else:
        print("你输入的是一个奇数")
```

2.3.2 for循环与while循环嵌套

已知 lst = [2, 3, 4]
依次要求用户输入2，3，4 的整数倍，先让用户输入2的倍数，如果用户输入的正确，输出“输入正确”，否则输出 “输入错误”，如果用户输入quit，则停止当前的输入，让用户输入3的倍数，输入3的倍数的过程中，如果用户输入quit，则让用户输入4的倍数

```
lst = [2, 3, 4]
for item in lst:
    while True:
        input_str = input("请输入{number}的倍数,想停止输入时，输入quit:".format(numbe
r=item))
        if input_str == 'quit':
```

```
            break
        number = int(input_str)
        if number % item == 0:
            print("输入正确")
        else:
            print("输入错误")
```

2.3.3 continue的好处

break是跳出循环体，continue是跳过continue语句后面的代码块，循环并不停止

题目要求：
使用input函数接受用户的输入，如果用户输入的数值小于等于10，则判断是奇数还是偶数，如果数值大于10，则输出“输入大于10，不判断奇偶”,用户输入quit，结束程序

```
while True:
    input_str = input("请输入一个正整数,如果想停止程序，输入quit:")
    if input_str == 'quit':
        break
    number = int(input_str)
    if number > 10:
        continue

    if number % 2 == 0:
        print("输入为偶数")
    else:
        print("输入为奇数")
```

当number大于10 的时候，后面的那4行代码就不会被执行，直接进入到下一次循环。

上面的代码，也可以不使用continue

```
while True:
    input_str = input("请输入一个正整数,如果想停止程序，输入quit:")
    if input_str == 'quit':
        break
    number = int(input_str)
    if number < 10:
        if number % 2 == 0:
            print("输入为偶数")
        else:
            print("输入为奇数")
```

两段代码，实现了一样的功能，但对比一下不难发现，使用了不使用continue，代码的嵌套层次更深，如果嵌套多了，会让代码变得难以阅读，难以管理

但使用continue，就可以减少代码层次，代码的理解和管理都更容易，大于10的时候，continue跳过后面的代码，在逻辑思考时，这种一刀两断的方法让思路更清晰。

3. 内置函数篇

本篇的练习题不是考察你如何使用python的内置函数，而是通过实现与内置函数相同功能的函数来达到锻炼提升编码能力的目的

3.1 abs

题目要求

abs函数返回数字的绝对值，请实现下面的函数，模仿abs函数的功能,返回数字的绝对值

```
def my_abs(number):
    pass
```

思路分析

处于程序健壮性考虑，要对传入的number参数进行检查，判断其类型是否为数字类型，float和int是比较常用的数据类型，复数类型基本接触不到，因此不考虑。

判断变量类型，可以使用isinstance函数，该函数的第一个参数是需要检查类型的对象，第二个参数可以是数据类型，也可以是一个元组，元组里是多个数据类型，只要满足其中一个就返回True

如果number的数值小于0，乘以-1就得到了绝对值

示例代码

```
def my_abs(number):
    if not isinstance(number, (float, int)):
        return number

    if number < 0:
        number *= -1

    return number
```

```
if __name__ == '__main__':
    print(my_abs(-3))
    print(my_abs(-3.9))
    print(my_abs(54.3))
```

3.2 sum

题目要求

sum函数可以获取列表所有数据的总和，模仿这个功能实现下面的函数，

```
def my_sum(lst):
    """
    返回列表里所有数据的总和
    如果列表里有非数字类型的数据，忽略不管
    :param lst:
    :return:
    """
    pass
```

思路分析

- 对传入的参数lst，要进行类型检查
- 遍历列表，遇到数字类型的数据就进行加和操作

示例代码

```
def my_sum(lst):
    """
    返回列表里所有数据的总和
    :param lst:
    :return:
    """
    sum_res = 0
    if not isinstance(lst, list):
        return sum_res

    for item in lst:
        if isinstance(item, (float, int)):
```

```
            sum_res += item

    return sum_res


if __name__ == '__main__':
    lst = [3, 4, '43', 5.4]
    print(my_sum(lst))
```

3.3 max

max函数返回序列中的最大值，传入的参数可以是列表，也可以是元组，实现下面的函数，实现同样的功能，如果序列里有非数字类型的数据，可以忽略,如果序列是空的，可以直接返回None

```
def my_max(seq):
    """
    返回序列里的最大值
    :param lst:
    :return:
    """
```

思路分析

对传入的参数seq要进行类型检查，如果既不是列表，也不是元组，那么就返回None

如果序列是空的，也可以直接返回None

遍历序列里的元素，如果数据的类型不属于数字类型，那么就忽略该数据

示例代码

```
def my_max(seq):
    """
    返回序列里的最大值
    :param lst:
    :return:
    """
    max_value = None
    if not isinstance(seq, (list, tuple)):
        return max_value
```

```
    if len(seq) == 0:
        return max_value

    max_value = seq[0]
    for item in seq:
        if not isinstance(item, (float, int)):
            continue
        if item > max_value:
            max_value = item

    return max_value


if __name__ == '__main__':
    lst = [3, 4, '43', 5.4]
    print(my_max(lst))
```

3.4 min

min函数返回序列中的最小值，传入的参数可以是列表，也可以是元组，实现下面的函数，实现同样的功能，如果序列里有非数字类型的数据，可以忽略

```
def my_min(seq):
    """
    返回序列里的最小值
    :param lst:
    :return:
    """
    pass
```

思路分析

整体思路与3.3的my_max函数相同

示例代码

```
def my_min(seq):
    """
    返回序列里的最小值
    :param lst:
    :return:
    """
```

```
    min_value = None
    if not isinstance(seq, (list, tuple)):
        return min_value

    if len(seq) == 0:
        return min_value

    min_value = seq[0]
    for item in seq:
        if not isinstance(item, (float, int)):
            continue

        if item < min_value:
            min_value = item

    return min_value


if __name__ == '__main__':
    lst = [3, 4, '43', 5.4]
    print(my_min(lst))
```

3.5 int

内置函数int，可以将float,全是数字的字符串转成int类型的数据，为了降低难度，这个练习题只要求你实现其中一种功能，将全是由数字组成的字符串转成int类型数据，例如将字符串"432" 转成整数432，函数定义如下

```
def my_int(string):
    """
    将字符串string转成int类型数据
    不考虑string的类型,默认就是符合要求的字符串
    传入字符串"432" 返回整数432
    :param string:
    :return:
    """
    pass
```

思路分析

题目的要求非常明确，只将"432"这种全是由数字组成的字符串转成int类型数据，这样就没什么难度了

遍历字符串，每个将字符串里的每个字符转成int类型的数值，这个过程可以使用字典来完成，建立一个字典，字符串的数字做key，int类型的数字做value,例如下面的字典

```
str_int_dic = {
    '0': 0,
    '1': 1,
    '2': 2,
    '3': 3,
    '4': 4,
    '5': 5,
    '6': 6,
    '7': 7,
    '8': 8,
    '9': 9
}
```

得到数字后，还得考虑这个数字是哪一位的，是千位还是百位，这里可以使用一个技巧，遍历的过程是从左向右进行的，设置一个变量保存转换后的int数据，初始值赋为0，每一次循环后，都用这个变量乘10再加上所遍历到数值，这样就巧妙的解决了位数问题。

示例代码

```
str_int_dic = {
    '0': 0,
    '1': 1,
    '2': 2,
    '3': 3,
    '4': 4,
    '5': 5,
    '6': 6,
    '7': 7,
    '8': 8,
    '9': 9
}

def my_int(string):
    """
    将字符串string转成int类型数据
    不考虑string的类型,默认就是符合要求的字符串
    传入字符串"432" 返回整数432
    :param string:
    :return:
    """
```

```
    res = 0
    for item in string:
        int_value = str_int_dic[item]
        res = res*10 + int_value

    return res

if __name__ == '__main__':
    print(my_int('432'))
```

3.6 str

题目要求

内置函数str的功能非常强大，想要模仿实现一个相同功能的函数是非常困难的，因此本练习题只要求你将int类型的数据转换成字符串，实现下面的函数

```
def my_str(int_value):
    """
    将int_value转换成字符串
    :param int_value:
    :return:
    """
    pass
```

思路分析

int类型的数据，不能像字符串那样使用for循环进行遍历，但可以结合 / 和 % 操作符从个位向高位进行遍历，获取到某一位的数字之后，将其转换成字符串，append到一个列表中。

遍历结束之后，翻转列表，用空字符串join这个列表，即可得到转换后的字符串。

单个数字，如何转成字符串呢？可以使用3.6中类似的方法，创建一个字典，数字为key，字符串数字为value

```
int_str_dict = {
    0: '0',
    1: '1',
    2: '2',
    3: '3',
    4: '4',
    5: '5',
```

```
    6: '6',
    7: '7',
    8: '8',
    9: '9',
}
```

获得某一位数字后，通过字典获得对应的字符串，此外，还可以通过ascii码表来获得与之对应的数字字符。以3为例，chr(3+48)即可得到字符串'3'，其原理，字符串3的ascii码表十进制数值为51，恰好比3大48，其他数值，也同样如此。

大致的思路已经清晰了，接下来是一些细节问题

- 如果传入的参数是0，那么直接返回字符串'0'
- 如果传入的参数是负数，需要标识记录，最后在列表里append一个'-' 字符串。
- lst = [1, 2, 3],想要翻转，lst = lst[::-1]

示例代码

```
def my_str(int_value):
    """
    将int_value转换成字符串
    :param int_value:
    :return:
    """
    if int_value == 0:
        return '0'

    lst = []
    is_positive = True
    if int_value < 0:
        is_positive = False
        int_value = abs(int_value)

    while int_value:
        number = int_value%10
        int_value //= 10
        str_number = chr(number+48)
        lst.append(str_number)

    if not is_positive:
        lst.append('-')

    lst = lst[::-1]
    return ''.join(lst)
```

```
if __name__ == '__main__':
    print(my_str(0))
    print(my_str(123))
    print(my_str(-123))
```

3.7 float

题目要求

为了降低难度，本题目只要求你将字符串转换成float类型的数据，且字符串都是符合”xx.xx“格式的字符串，例如"34.22"

```
def my_float(string):
    """
    将字符串string转换成float类型数据
    :param string:
    :return:
    """
    pass
```

题目分析

使用split函数，以"."做分隔符，可以将字符串分割为两部分，整数部分和小数部分，这两个部分可以分别用3.5 中的my_int 函数进行处理，以"34.22"为例，分别得到整数34 和22，对于22，不停的乘以0.1，知道它的数值小于1，就得到了小数部分

示例代码

```
str_int_dic = {
    '0': 0,
    '1': 1,
    '2': 2,
    '3': 3,
    '4': 4,
    '5': 5,
    '6': 6,
    '7': 7,
    '8': 8,
    '9': 9
```

```
}

def my_int(string):
    """
    将字符串string转成int类型数据
    不考虑string的类型,默认就是符合要求的字符串
    传入字符串"432" 返回整数432
    :param string:
    :return:
    """
    res = 0
    for item in string:
        int_value = str_int_dic[item]
        res = res*10 + int_value

    return res


def my_float(string):
    """
    将字符串string转换成float类型数据
    :param string:
    :return:
    """
    arrs = string.split('.')
    int_value = my_int(arrs[0])
    float_value = my_int(arrs[1])
    while float_value > 1:
        float_value *= 0.1

    return int_value + float_value


if __name__ == '__main__':
    print(my_float("34.22"))
```

3.8 len

题目要求

内置函数可以获得可迭代对象的长度，例如字符串，列表，元组，字典，集合。实现一个类似功能的函数，获得数据的长度。

```
def my_len(obj):
    """
    获得obj对象的长度
    :param obj:
    :return:
    """
    pass
```

思路分析

使用for循环遍历对象，循环的次数就是这个对象的长度，只需要一个变量来保存循环的次数就可以了。

对obj参数的检查，可以使用isinstance判断是否为列表，元组，字典，集合，字符串中的某一个，更为简便的做法，这些对象都是可迭代对象，isinstance(obj, Iterable) 可以判断obj是否为可迭代对象

示例代码

```
from collections import Iterable


def my_len(obj):
    """
    获得obj对象的长度
    :param obj:
    :return:
    """
    if not isinstance(obj, Iterable):
        return None

    length = 0
    for item in obj:
        length += 1

    return length


if __name__ == '__main__':
    print(my_len('232'))
    print(my_len([3, 4, 2, 1]))
    print(my_len({'a': 4, 'b': 4}))
    print(my_len((3, 5, 6, 6, 3)))
```

```
    print(my_len(set([3, 5, 6, 6, 3])))
```

3.9 enumerate

题目要求

enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中，下面是使用示例

```
lst = ['a', 'b', 'c']
for index, item in enumerate(lst):
    print(index, item)
```

程序输出

```
0 a
1 b
2 c
```

请仿造该功能实现下面的函数

```
def my_enumerate(lst):
    """
    实现和enumerate 类似的功能
    :param lst:
    :return:
    """
    pass
```

思路分析

想要实现这个函数，只需两行代码就可以了,不过，这需要你对生成器有一定的理解和认识。一个函数里如果出现了yield关键字，那么这个函数就是生成器函数，该函数返回的是一个生成器。

yield有着和return相似的功能，都会将数据返回给调用者，不同之处在于，return执行后，函数结束了，而yield执行后，会保留当前的状态，等到下一次执行时，恢复之前的状态，继续执行。

在函数内部，使用for循环通过索引

遍历lst, 使用yield返回索引和索引位置上的元素。

示例代码

```
def my_enumerate(lst):
    """
    实现和enumerate 类似的功能
    :param lst:
    :return:
    """
    for i in range(len(lst)):
        yield i, lst[i]


lst = ['a', 'b', 'c']
for index, item in my_enumerate(lst):
    print(index, item)
```

3.10 all

题目要求

all() 函数用于判断给定的可迭代参数 iterable 中的所有元素是否都为 True，示例代码如下

```
lst = [True, False, True]
print(all(lst))
```

最终输出结果是False,实现下面的函数，完成类似的功能

```
def my_all(seq):
    """
    如果列表里所有的元素都是True,则函数返回True,反之,返回False
    :param seq: 列表
    :return:
    """
    pass
```

为了简化难度，参数seq默认只传列表

思路分析

老规矩，使用for循环遍历列表，当前遍历到的元素如果是False，直接返回False

示例代码

```
def my_all(seq):
    """
    如果列表里所有的元素都是True,则函数返回True,反之,返回False
    :param seq: 列表
    :return:
    """
    for item in seq:
        if not item:
            return False

    return True

if __name__ == '__main__':
    print(my_all([True, False, True]))
```

3.11 any

题目要求

any函数用于判断给定的可迭代参数 iterable 中的所有元素是否至少有一个为True
示例代码

```
lst = [True, False, False]
print(any(lst))
```

输出结果为True
实现下面的函数，完成类似的功能，默认传入的参数是列表

```
def my_any(lst):
    """
    列表lst中有一个True,则函数返回True
    :param lst:
    :return:
    """
    pass
```

思路分析

老规矩，遍历列表，当前遍历到的元素如果为True，则函数返回True

示例代码

```
def my_any(lst):
    """
    列表lst中有一个True,则函数返回True
    :param lst:
    :return:
    """
    for item in lst:
        if item:
            return True

    return False

if __name__ == '__main__':
    print(my_any([True, False, False]))
```

3.12 bin

函数bin可以获得整数的二进制形式，示例代码如下

```
print(bin(10))
```

程序输出结果为0b1010。

实现下面的函数，完成相同的功能，为了降低难度，你的算法只需要考虑正整数，而且二进制前面不需要加0b

```
def my_bin(value):
    """
    返回正整数value的二进制形式
    :param value:
    :return:
    """
```

思路分析

在算法面试中，有一道题目经常被使用，它要求应聘者计算一个整数的二进制中1的个数。解决的思路是判断二进制最后一位是否为1，如果为1，则计数器加1，判断完成后，整数向右位移一位（使用位运算符 >>），继续判断二进制的最后一位是否为1.

本练习题可以采用相同的思路

示例代码

```
def my_bin(value):
    """
    返回正整数value的二进制形式
    :param value:
    :return:
    """
    lst = []
    while value:
        if value % 2 == 1:
            lst.append('1')
        else:
            lst.append('0')

        value = value >>1

    lst = lst[::-1]
    return ''.join(lst)


print(bin(10))

if __name__ == '__main__':
    print(my_bin(3))
    print(my_bin(8))
    print(my_bin(10))
```

4. 字符串方法

4.1 find

```
def my_find(source, target, start=0):
    """
    返回字符串source中 子串target开始的位置， 从start索引开始搜索
```

```
    如果可以找到多个，返回第一个
    :param source:
    :param target:
    :param start:
    :return:
    """
    if not source or not target:
        return -1

    # 不合理的搜索起始位置
    if start < 0 or start >= len(source):
        return -1

    if len(target) > len(source):
        return -1

    for index in range(start, len(source) - len(target)+1):
        t_index = 0
        while t_index < len(target):
            if target[t_index] == source[t_index+index]:
                t_index += 1
            else:
                break

        if t_index == len(target):
            return index

    return -1


if __name__ == '__main__':
    print(my_find('this is a book', 'this'))
    print(my_find('this is a book', 'this', start=1))
    print(my_find('this is a book', 'book'))
    print(my_find('this is a book', 'k', start=10))
    print(my_find('this is a book', 'book', start=10))
    print(my_find('this is a book', 'a', start=3))
```

4.2 replace

```
def my_replace(source, oldsub, newsub):
    """
    将字符串里所有的oldsub子串替换成newsub
    :param source:
```

```
    :param old:
    :param new:
    :return:
    """
    if not source or not oldsub:
        return source

    new_string = ""
    start_index = 0
    index = my_find(source, oldsub, start=start_index)
    while index != -1:
        new_string += source[start_index:index] + newsub
        start_index = index+len(oldsub)
        index = my_find(source, oldsub, start=start_index)

    new_string += source[start_index:]
    return new_string


def my_find(source, target, start=0):
    """
    返回字符串source中 子串target开始的位置， 从start索引开始搜索
    如果可以找到多个，返回第一个
    :param source:
    :param target:
    :param start:
    :return:
    """
    if not source or not target:
        return -1

    # 不合理的搜索起始位置
    if start < 0 or start >= len(source):
        return -1

    if len(target) > len(source):
        return -1

    for index in range(start, len(source) - len(target)+1):
        t_index = 0
        while t_index < len(target):
            if target[t_index] == source[t_index+index]:
                t_index += 1
            else:
                break
```

```
        if t_index == len(target):
            return index

    return -1


if __name__ == '__main__':
    print(my_replace('this is a book', 'this', 'it'))
    print(my_replace('this is a this book', 'this', 'it'))
    print(my_replace('this is a this bookthis', 't2his', 'it'))
```

4.3 split

```
def my_find(source, target, start=0):
    """
    返回字符串source中 子串target开始的位置， 从start索引开始搜索
    如果可以找到多个，返回第一个
    :param source:
    :param target:
    :param start:
    :return:
    """
    if not source or not target:
        return -1

    # 不合理的搜索起始位置
    if start < 0 or start >= len(source):
        return -1

    if len(target) > len(source):
        return -1

    for index in range(start, len(source) - len(target)+1):
        t_index = 0
        while t_index < len(target):
            if target[t_index] == source[t_index+index]:
                t_index += 1
            else:
                break

        if t_index == len(target):
            return index

    return -1
```

```
def my_split(source, sep, maxsplit=-1):
    """
    以sep分割字符串source
    :param source:
    :param sep:
    :param maxsplit:
    :return:
    """
    if not source or not sep:
        return []

    lst = []
    max_split_count = maxsplit if maxsplit >0 else len(source)
    split_count = 0

    start_index = 0
    index = my_find(source, sep, start=start_index)
    while split_count < max_split_count and index != -1:
        sep_str = source[start_index:index]
        lst.append(sep_str)
        split_count += 1
        start_index = index + len(sep)
        index = my_find(source, sep, start=start_index)

    sep_str = source[start_index:]
    lst.append(sep_str)

    return lst


if __name__ == '__main__':
    print(my_split("1,3,4", ','))
    print(my_split("1,,3,,4", ',,'))
    print(my_split("abcadae", 'a'))
    print(my_split("abcadae", 'a', maxsplit=2))
    print(my_split("aaaa", 'a'))
```

4.4 字符串大写转小写

题目要求

实现函数

```
def lower(string):
    """
    将字符串string里所有的大写字母改成小写字母，并返回一个新的字符串
    :param string:
    :return:
    """
```

思路分析

实现大小写转换，首先要能识别出一个字符是否为大写字母，你可以在得到这个字符后，判断其是否在A和Z之间，更专业的办法是通过ord 函数获得这个字符的ASCII码表的十进制数值，判断其是否在65和90之间。

获得字符的ASCII码表的十进制数值，其目的不仅仅是判断它是否为大写字母，第二个目的是通过这个十进制数值与32相加，来获得大写字母所对应的小写字母的十进制数值，这样，才能准确的转换成小写字母。

我在程序里使用list函数将字符串转成列表，之所以这样做，是因为字符串是不可变类型的数据，无法直接修改，只好先将其转成列表，将列表里的大写字母转成小写字母，再将列表转成字符串。

示例代码

```
def lower(string):
    """
    将字符串string里所有的大写字母改成小写字母，并返回一个新的字符串
    :param string:
    :return:
    """
    if not string:
        return None

    lst = list(string)
    for index, item in enumerate(lst):
        ascii_index = ord(item)
        if 65 <= ascii_index <= 90:
            s = chr(ascii_index+32)
            lst[index] = s

    return ''.join(lst)
```

```
if __name__ == '__main__':
    print(lower('232rSFD'))
```

4.5 判断字符串是否全部为小写字母

题目要求

实现函数

```
def islower(string):
    """
    如果字符串string 里所有区分大小写的字符都是小写，则返回True
    :param string:
    :return:
    """
    pass
```

比如传入字符串 "iwj32as"，函数应该返回True

思路分析

字符串里，常见的只有26个英文字母是区分大小写的，因为，咱们只关心英文字母即可。

遍历字符串，逐个字符进行检查，获得其ASCII码表里的十进制数值，如果该数值在65到90之间，一定是大写字母，此时返回False，如果for循环结束后，仍然没有返回False，那么就说明，字符串里没有大写字母，可以返回True

示例代码

```
def islower(string):
    """
    如果字符串string 里所有区分大小写的字符都是小写，则返回True
    :param string:
    :return:
    """
    if not string:
        return False

    for item in string:
        if 65 <= ord(item) <= 90:
```

```
            return False

    return True


if __name__ == '__main__':
    print(islower('232r'))
```

4.6 实现isdigit

题目要求

实现函数isdigit， 判断字符串里是否只包含数字0~9

```
def isdigit(string):
    """
    判断字符串只包含数字
    :param string:
    :return:
    """
    pass
```

思路分析

遍历字符串，对每个字符做检查，如果都是0到9的某个数值，那么函数返回True，只要有一个不是0到9，就返回False。

如何确定一个字符是不是0到9中的某一个呢，方法很多，你可以用if条件判断语句判断字符是否在列表['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']中，也可以像我下面示例代码一样，使用ord函数获得字符的ASCII编码对应的10进制数值，接着判断是否在48到57之间。

示例代码

```
def isdigit(string):
    """
    判断字符串只包含数字
    :param string:
    :return:
    """
    if not string:
        return False
```

```
    for item in string:
        if not (48 <= ord(item) <= 57):
            return False

    return True


if __name__ == '__main__':
    print(isdigit('232'))
    print(isdigit('232r'))
    print(isdigit(''))
```

4.7 startswith

题目要求

实现函数is_startswith，如果字符串source是以substr开头的，则函数返回True,反之返回False

```
def is_startswith(source, substr):
    """
    判断字符串source是否以substr开头
    :param source:
    :param substr:
    :return:
    """
    pass
```

思路分析

函数首先要判断传入的参数是否合法，这里默认传入的都是字符串，那么我们要需要判断字符串是否有空串的情况

如果substr的长度大于source的长度，直接返回False

从索引0开始，遍历substr,从source上获得相同索引的字符，两者进行比较，只要有一个字符不相同，则可以立即返回False

示例代码

```
def is_startswith(source, substr):
    """
    判断字符串source是否以substr开头
    :param source:
    :param substr:
    :return:
    """
    if not source or not substr:
        return False

    if len(substr) > len(source):
        return False

    for index, item in enumerate(substr):
        if item != source[index]:
            break
    else:
        return True  # 如果for循环不是因为break结束的，就会进入到else语句块

    return False


if __name__ == '__main__':
    print(is_startswith("python", 'py'))
```

4.8 endswith

题目要求

实现函数is_endswith，判断字符串source是否以substr结尾

```
def is_endswith(source, substr):
    """
    判断字符串source 是否以substr结尾
    :param source:
    :param substr:
    :return:
    """
    pass
```

思路分析

这个练习题的解法其实和is_startswith函数相差无几，所不同的是，在is_startswith函数中，要从索引0开始进行相同位置字符的比较，而现在，是要判断是否以substr结尾，所以我们从索引len(source) – len(substr)开始逐一进行比较

示例代码

```
def is_endswith(source, substr):
    """
    判断字符串source 是否以substr结尾
    :param source:
    :param substr:
    :return:
    """
    if not source or not substr:
        return False

    if len(substr) > len(source):
        return False

    start_index = len(source) - len(substr)
    for index in range(start_index, len(source)):
        if source[index] != substr[index-start_index]:
            break
    else:
        return True

    return False

if __name__ == '__main__':
    print(is_endswith("python", 'thon'))
```

4.9 实现字符串capitalize方法

题目要求

capitalize方法将字符串的第一个字母转成大写，其他字母转成小写，请实现函数my_capitalize，完成同样的功能

```
def my_capitalize(string):
    pass
```

思路分析

遍历字符串，如果首字母是小写，则转成大写，其余索引上的字母如果是大写，则转成小写

大小写转换的方法，可以参考lower函数的实现

示例代码

```
def my_capitalize(string):
    if not string:
        return string

    lst = []
    for index, item in enumerate(string):
        ascii_index = ord(item)
        if index == 0:
            if 97 <= ascii_index <= 122:
                item = chr(ascii_index-32)
        else:
            if 65 <= ascii_index <= 90:
                item = chr(ascii_index+32)
        lst.append(item)

    return "".join(lst)


print(my_capitalize('this is A book'))
```

4.10 实现字符串count方法

题目要求

字符串count方法，可以返回指定范围内的子串的数量，下面是用法示例

```
source = "this is a book"
target = 'is'

print(source.count(target))
```

程序输出2

请仿照字符串的count方法，实现下面的函数

```
def my_count(source, target, start, end):
    """
    函数返回字符串source在start 和 end之前，子串target 的数量， 索引范围左闭右开
    :param source:
    :param target:
    :param start:
    :param end:
    :return:
    """
    pass
```

思路分析

对于传入的参数进行合法性判断，是编写函数时必须要考虑的事情

- 字符串source，target为空判断
- start >= end判断
- start范围判断，不能小于0，也不能大于等于len(source)
- 如果end大于len(source)，则重新对end赋值，让其等于len(source)，算是一种合理的补救

经过前面的4个判断后，基本上可以保证传入的参数是合法的，不至于因为参数不合法导致程序出错

具体实现思路，将字符串target索引0与字符串source索引start进行对齐，然后逐个字符比较，只要有一个不相同，就说明，source[start: len(target)] != target，那么就需要向右移动一位，比较source[start+1: len(target)]是否与target相等。

代码里最精髓的是if t_index == len(target)这行，如果对比过程中，触发了break，那么t_index一定不会与len(target)相等，就依靠这个条件判断，就可以知道是不是找到了子串。

示例代码

```
def my_count(source, target, start, end):
    """
    函数返回字符串source在start 和 end之前，子串target 的数量， 索引范围左闭右开
    :param source:
    :param target:
    :param start:
    :param end:
    :return:
    """
    if not source or not target:
```

```
        return 0

    if start >= end:
        return 0

    if start >= len(source) or start < 0:
        return 0

    count = 0
    if end > len(source):
        end = len(source)

    index = start

    while index < end:
        t_index = 0
        while t_index < len(target) and index+len(target) <= end:
            if target[t_index] != source[index+t_index]:
                break
            t_index += 1

        if t_index == len(target):
            index += len(target)
            count += 1
        else:
            index += 1

    return count


source = "this is a book"
target = 'is'

print(my_count(source, target, 0, len(source)))
```

5. 排序算法篇

排序算法最能体现一个程序员的算法功底，也是面试时经常被拿来考察候选者的题目，本篇章一共讲解8种排序算法。

5.1 冒泡排序

冒泡排序的核心思想是相邻的两个数据进行比较，假设数列A有n个数据，先比较第1个和第2

个数据，如果A1 > A2,则交换他们的位置，确保较大的那个数在右侧。

接下来比较A2和A3，采用相同的规则，较大的数向右移动，最后会比较An−1 和An的大小，如果An−1 > An，那么交换他们的位置，这时，An是数列中的最大值。

你肯定已经发现，经过这一轮比较后，数列仍然是无序的，但是没有关系，我们已经找到了最大值An，而且它在队列的末尾。

接下来要做的事情，就是简单的重复之前的过程，整个数列，先暂时把An排除在外，这n−1个无序的数，仍然可以采用之前的方法，找出n−1个数当中的最大值，这样An−1就是第2大的数，继续对n−2个数做相同的事情

为了让你更容易理解冒泡排序，我们先实现一个简单的函数

```
def move_max(lst, max_index):
    """
    将索引0到max_index这个范围内的最大值移动到max_index位置上
    :param lst:
    :param max_index:
    :return:
    """
    for i in range(max_index):
        if lst[i] > lst[i+1]:
            lst[i], lst[i+1] = lst[i+1], lst[i]


if __name__ == '__main__':
    lst = [7, 1, 4, 2, 3, 6]
    move_max(lst, len(lst)-1)
    print(lst)
```

这个函数只完成一个简单的功能，它将列表从索引0到max_index之间的最大值移动max_index索引上，这正是冒泡排序的核心思想。

当我们完成这一步，剩下的事情，就是不断的重复这个过程。

```
def pop_sort(lst):
    for i in range(len(lst)-1, 1, -1):
        move_max(lst, i)


def move_max(lst, max_index):
    """
    将索引0到max_index这个范围内的最大值移动到max_index位置上
```

```
    :param lst:
    :param max_index:
    :return:
    """
    for i in range(max_index):
        if lst[i] > lst[i+1]:
            lst[i], lst[i+1] = lst[i+1], lst[i]


if __name__ == '__main__':
    lst = [7, 1, 4, 2, 3, 6]
    pop_sort(lst)
    print(lst)
```

5.2 快速排序

快速排序的思路可以归结为3个步骤

1. 从待排序数组中随意选中一个数值，作为基准值
2. 移动待排序数组中的元素，是的基准值左侧的数值都小于等于它，右侧的数值大于等于它
3. 基准值将原来的数组分为两部分，针对这两部分，重复步骤1，2， 3

通过分析，可以确定，必然要使用递归算法，遇到递归不要害怕，先把1，2两步分区实现，最后实现第3步的递归

先实现分区

```
def partition(lst,start,end):
    """
    用lst[start] 做基准值，在start到end这个范围进行分区
    """
    pivot = lst[start]

    while start < end :
        while start < end and lst[end] >= pivot:
            end -= 1
        lst[start] = lst[end]

        while start < end and lst[start] <= pivot:
            start += 1
        lst[end] = lst[start]

    lst[start] = pivot
```

```
    return start
```

partition函数返回基准值最后的索引，知道这个索引，才能将之前的待排序数组分为两部分，下面是递归部分的实现

```
def my_quick_sort(lst,start,end):
    if start>= end:
        return

    index = partition(lst,start,end)
    my_quick_sort(lst,start,index-1)
    my_quick_sort(lst,index+1,end)
```

虽然这两段代码里的逻辑，你还有些不清楚，但整个的分析过程应该说是比较清晰的，先实现分区，然后实现递归，在编写算法时，很忌讳大包大揽的考虑问题，不分层次，不分先后，不分轻重。

分区虽然没有让整个数组变得有序，但是让基准值找到了自己应该在的位置，对左右两侧重复分区动作，每一次分区动作都至少让一个元素找到自己应该在的位置。

验证代码

```
if __name__ == '__main__':
    lst = [4,3,2,4,1,5,7,2]
    my_quick_sort(lst,0,len(lst)-1)
    print lst
```

5.3 希尔排序

算法定义

希尔排序，又称缩小增量排序，不要被这个名字吓到，其实，它只是对插入算法的改进而已。

当待排序列基本有序的情况下，插入算法的效率非常高，那么希尔排序就是利用这个特点对插入算法进行了改造升级

分组

待排序数组为

```
4,1,67,34,12,35,14,8,6,19
```

第一轮希尔排序

希尔排序的关键在于对待排序列进行分组，这个分组并不是真的对序列进行了拆分，而仅仅是虚拟的分组

首先，用10/2 = 5, 这里的5就是缩小增量排序中的那个“增量”。从第0个元素开始，每个元素都与自己距离为5的元素分为一组，那么这样一来分组情况就是

```
4   35
1   14
67  8
34  6
12  19
```

需要注意的是，所谓的分组，仅仅是逻辑上的分组，这10个元素仍然在原来的序列中。上面一共分了5组，每一组都进行插入排序，67 和 8 交换位置，34 和6 交换位置，这样第一次分组后并对各组进行插入排序后，序列变成了

```
4, 1, 8, 6, 12, 35, 14, 67, 34, 19
```

第二轮希尔排序

上一轮排序时，增量为5，那么这一轮增量为5/2 = 2，这就意味着，从第0个元素开始，每个元素都与自己距离为2的元素分为一组，分组情况如下

```
4 8 12 14 34
1 6 35 67 19
```

整个序列被分成了两组，分别对他们进行插入排序，排序后的结果为

```
4, 1, 8, 6, 12, 19, 14, 35, 34, 67
```

第三轮希尔排序

上一轮排序时，增量为2，这一轮增量为2 /2 = 1，当增量为1的时候，其实就只能分出一个组了，这样，就完全的退化成插入排序了，但是，由于已经进行了两轮希尔排序，使得序列已经

基本有序了，那么此时进行插入排序，效果就会非常好

增量从5变为2，从2变为1，是逐渐减小的过程，增量是分组时所使用的步长。

示例代码

有了前面的概念以及算法的理解，写出代码就变得容易了，先分组，然后进行插入排序，你唯一要注意的地方是进行插入排序时，要弄清楚，哪些元素是一组的

```
lst = [4,1,67,34,12,35,14,8,6,19]
length = len(lst)
step = length//2

while step > 0:

    for i in range(step):
        # 插入排序
        for j in range(i+step, length, step):
            if lst[j] < lst[j-step]:
                tmp = lst[j]
                k = j-step

                while k >= 0 and lst[k] > tmp:
                    lst[k+step] = lst[k]
                    k -= step

                lst[k+step] = tmp
    step //= 2 #缩小增量

print lst
```

5.4 归并排序

合并两个有序集合

有两个有序的序列，分别为 [1,4,7] ,[2,3,5],现在请考虑将这两个序列合并成一个有序的序列。

其实方法真的非常简单

1. 首先创建一个新的序列，分别从两个序列中取出第一个数，1和2，1比2小，把1放到新的序列中

2. 第一个序列中的1已经放到新序列中，那么拿出4来进行比较，2比4小，把2放到新的序列中
3. 第二个序列中的2已经放到新序列中，那么拿出3来进行比较，3比4小，把3放到新的序列中
4. 第二个序列中的3已经放到新序列中，那么拿出5来进行比较，4比5小，把4放到新的序列中
5. 第一个序列中的4已经放到新序列中，那么拿出7来进行比较，5比7小，把5放到新的序列中
6. 最后把7放入到新的序列中

合并的方法就是分别从两个序列中拿出一个数来进行比较，小的那一个放到新序列中，然后，从这个小的数所属的序列中拿出一个数来继续比较

示例代码

```
def merge_lst(left_lst,right_lst):
    left_index, right_index = 0, 0
    res_lst = []
    while left_index < len(left_lst) and right_index < len(right_lst):
        # 小的放入到res_lst中
        if left_lst[left_index] < right_lst[right_index]:
            res_lst.append(left_lst[left_index])
            left_index += 1
        else:
            res_lst.append(right_lst[right_index])
            right_index += 1

    # 循环结束时,必然有一个序列已经都放到新序列里,另一个却没有
    if left_index == len(left_lst):
        res_lst.extend(right_lst[right_index:])
    else:
        res_lst.extend(left_lst[left_index:])

    return res_lst
```

5.5 归并排序

归并排序，利用了合并有序序列的思想，把一个序列分成A,B两个序列，如果这两个序列是有序的，那么直接合并他们不就可以了么，但是A，B两个序列未必是有序的，没关系，就拿A序

列来说，我把A序列再分一次，分成A1,A2，如果A1，A2有序我直接对他们进行合并，A不就变得有序了么，但是A1，A2未必有序啊，没关系，我继续分，直到分出来的序列里只有一个元素的时候，一个元素，就是一个有序的序列啊，这个时候不就可以合并了

这样一层一层的分组，分到最后，一个组里只有一个元素，终于符合合并的条件了，再一层一层的向上合并

完成示例代码：

```
def merge_lst(left_lst,right_lst):
    left_index, right_index = 0, 0
    res_lst = []
    while left_index < len(left_lst) and right_index < len(right_lst):
        # 小的放入到res_lst中
        if left_lst[left_index] < right_lst[right_index]:
            res_lst.append(left_lst[left_index])
            left_index += 1
        else:
            res_lst.append(right_lst[right_index])
            right_index += 1

    # 循环结束时,必然有一个序列已经都放到新序列里,另一个却没有
    if left_index == len(left_lst):
        res_lst.extend(right_lst[right_index:])
    else:
        res_lst.extend(left_lst[left_index:])

    return res_lst


def merge_sort(lst):
    if len(lst) <= 1:
        return lst

    middle = len(lst)//2
    left_lst = merge_sort(lst[:middle])
    right_lst = merge_sort(lst[middle:])
    return merge_lst(left_lst, right_lst)


if __name__ == '__main__':
    lst = [19,4,2,8,3,167,174,34]
    print merge_sort(lst)
```

5.6 选择排序

假设有一个序列，a[0],a[1],a[2]...a[n]现在，对它进行排序。我们先从0这个位置到n这个位置找出最小值，然后将这个最小值与a[0]交换，然后呢，a[1]到a[n]就是我们接下来要排序的序列

我们可以从1这个位置到n这个位置找出最小值，然后将这个最小值与a[1]交换，之后，a[2]到a[n]就是我们接下来要排序的序列

每一次，我们都从序列中找出一个最小值，然后把它与序列的第一个元素交换位置，这样下去，待排序的元素就会越来越少，直到最后一个

示例代码

```
def select_sort(lst):
    for i in range(len(lst)):
        min = i
        for j in range(min,len(lst)):
            # 寻找min 到len(lst)-1 这个范围内的最小值
            if lst[min] > lst[j]:
                min = j
        lst[i], lst[min] = lst[min], lst[i]

lst = [2,6,1,8,2,4,9]
select_sort(lst)
print lst
```

5.7 堆排序

堆的概念

堆是一种数据结构，分最大堆和最小堆，最大（最小）堆是一棵每一个节点的键值都不小于（大于）其孩子（如果存在）的键值的树。大顶堆是一棵完全二叉树，同时也是一棵最大树。小顶堆是一棵完全完全二叉树，同时也是一棵最小树。

我们用list来描述它

```
lst = [9, 6, 8, 3, 1, 4, 2]
```

你看不出这个lst有什么特别，别着急，再介绍两个概念给你

父节点，子节点

列表中的每一个元素都是一个节点，以lst[0] 为例，他的子节点分别是lst[1],lst[2]，同时我们也说lst[1]的父节点是lst[0]

我们可以计算每一个节点的子节点，假设当前节点的序号是i，那么它的左子节点则是 i*2 +1, 右子节点则是*i2 + 2

最大堆，最小堆

所谓最大堆就是指每一个节点的值都比它的子节点的值大，最小堆就是指每一个节点的值都比它的子节点的值小

现在，我们再来看上面给出的列表

lst[0] = 9,它的子节点分别是lst[1]=6,lst[2]=8

lst[1] = 6,它的子节点分别是lst[3]=3,lst[4]=1

lst[2] = 8,它的子节点分别是lst[5]=4,lst[6]=2

lst[3] = 3,它的子节点分贝是lst[7]和lst[8]，但这两个节点是不存在的

后面的也就不用再看了，这个列表符合最大堆的要求，父节点的值大于两个子节点的值，而且最重要的一点，堆中任意一颗子树仍然是堆

如何建立一个堆

关于堆的应用，非常多，比如堆排序，在应用之前，我们必须先建立一个堆，刚才给出的列表，恰好是一个堆，如果不是堆呢，我们需要将其变成堆，例如下面这个列表

```
lst = [3, 9, 2, 6, 1, 4, 8]
```

这个列表里的元素和上一个列表里的元素是一样的，只是顺序不同，建立堆的过程，就是调整顺序的过程，使其满足堆的定义

不是所有节点都有子节点

如果当前节点的位置是i,那么子节点的位置是i*2 + 1 和 i*2 +2 ，因此，不是所有节点都有子节点，假设一个堆的长度为n，那么n/2 – 1 及以前的节点都是有子节点的，这是一个非常简单的算数题，你稍微用脑就能理解。

那么建立堆的过程，就是从n/2 – 1 到0 逐渐调整的过程，如何调整呢?

每个节点都和自己的两个子节点中最大的那个节点交换位置就可以了，这样，节点值较大的那个就会不停的向上调整

示例代码

```
def adjust_heap(lst, i, size):
    lchild = 2 * i + 1      # 计算两个子节点的位置
    rchild = 2 * i + 2
    max = i
    if i < size // 2:
        if lchild < size and lst[lchild] > lst[max]:
            max = lchild
        if rchild < size and lst[rchild] > lst[max]:
            max = rchild

        # 如果max != i成立,那么就说明,子节点比父节点大,要交换位置
        if max != i:
            lst[max], lst[i] = lst[i], lst[max]
            # 向下继续调整,确保子树也符合堆的定义
            adjust_heap(lst, max, size)

def build_heap(lst):
    for i in range((len(lst)//2)-1, -1, -1):
        adjust_heap(lst, i, len(lst))
        print lst   # 此处输出,可以观察效果

lst = [3,9,2,6,1,4,8]
build_heap(lst)
print lst
```

最大堆，最小堆

关于最大堆，最小堆，我们只要掌握一点就好了，对于最大堆，堆定的元素一定是整个堆里最大的，但是，如果我们去观察，整个堆并不呈现有序的特性，比如前面建立的堆

```
[9, 6, 8, 3, 1, 4, 2]
```

堆顶元素为9，是最大值，但是从0到最后一个元素，并不是有序的

堆排序的思路

lst = [9, 6, 8, 3, 1, 4, 2]

（1）将lst[0] 与 lst[6]交换，交换后为[2, 8, 6, 4, 3, 1, 9],现在，这个堆已经被破坏掉了

（2）那么我们可以利用adjust_heap函数重新把它调整成一个堆，adjust_heap(lst,0,6)，这样调整后，lst=[8, 6, 4, 3, 1, 2, 9]

注意看lst[0]到lst[5],这个范围内的数据被调整成了一个堆，使得lst[0]也就是8是这个范围内的最大值

我们只需要重复刚才的两个步骤，就可以将堆的大小逐步缩小，同时，从后向前让整个lst变得有序

示例代码

```
def adjust_heap(lst, i, size):
    lchild = 2 * i + 1      # 计算两个子节点的位置
    rchild = 2 * i + 2
    max = i
    if i < size // 2:
        if lchild < size and lst[lchild] > lst[max]:
            max = lchild
        if rchild < size and lst[rchild] > lst[max]:
            max = rchild

        # 如果max != i成立,那么就说明,子节点比父节点大,要交换位置
        if max != i:
            lst[max], lst[i] = lst[i], lst[max]
            # 向下继续调整,确保子树也符合堆的定义
            adjust_heap(lst, max, size)

def build_heap(lst):
    for i in range((len(lst)//2)-1, -1, -1):
        adjust_heap(lst, i, len(lst))
        print lst   # 此处输出,可以观察效果


lst = [3,9,2,6,1,4,8]
def heap_sort(lst):
    size = len(lst)
    # 建立一个堆,此时lst[0]一定是最大值
    build_heap(lst)
    print '*'*20
    for i in range(size-1, -1, -1):
        # 将lst[0] 与lst[i] 交换,这样大的数值就按顺序排到了后面
        lst[0], lst[i] = lst[i], lst[0]
        # 交换后,重新将0到i这个范围内的数据调整成堆,这样lst[0]还是最大值
```

```
        adjust_heap(lst, 0, i)
        print lst,i  # 此处输出可以查看效果

heap_sort(lst)
print '*'*20
print lst
```

5.8 插入排序

插入排序就是每一步都将一个待排数据按其大小插入到已经排序的数据中的适当位置，直到全部插入完毕。

咱们举一个例子，你就能明白插入排序的精髓所在

```
lst = [1, 2, 6, 7, 5]
```

lst是一个待排序的列表，你仔细观察不难发现，这个列表里的前4个数据已经是有序的了，现在，只需要把最后一个元素5插入到一个合适的位置就可以了。

从7开始向左遍历，比5大的数向右移动，当遇到一个小于等于5的数就停下来，这个位置就是5应该在的位置。当7向右移动时，占据了5的位置，因此，程序里需要一个变量把5保存下来，还需要一个变量把向左遍历时的索引记录下来，最后这个索引就是5应该在的位置。

```
def insert(lst, index):
    """
    列表lst从索引0到索引index-1 都是有序的
    函数将索引index位置上的元素插入到前面的一个合适的位置
    :param lst:
    :param index:
    :return:
    """
    if lst[index-1] < lst[index]:
        return

    tmp = lst[index]
    tmp_index = index
    while tmp_index > 0 and lst[tmp_index-1] > tmp:
        lst[tmp_index] = lst[tmp_index-1]
        tmp_index -= 1
    lst[tmp_index] = tmp
```

```
if __name__ == '__main__':
    lst = [1, 2, 6, 7, 5]
    insert(lst, 4)
    print(lst)
```

函数默认列表lst从0到index-1都是有序的，现在只需要将索引index位置上的数据插入到合适的位置就可以了。

你可能已经产生了疑惑，咱们是要排序啊，你这函数默认前面index-1个数据都是有序的，这不合理啊，前面index-1个数据如果是无序的，你怎么办呢?

看下面的代码，你就全明白了

```
def insert(lst, index):
    """
    列表lst从索引0到索引index-1 都是有序的
    函数将索引index位置上的元素插入到前面的一个合适的位置
    :param lst:
    :param index:
    :return:
    """
    if lst[index-1] < lst[index]:
        return

    tmp = lst[index]
    tmp_index = index
    while tmp_index > 0 and lst[tmp_index-1] > tmp:
        lst[tmp_index] = lst[tmp_index-1]
        tmp_index -= 1
    lst[tmp_index] = tmp


def insert_sort(lst):
    for i in range(1, len(lst)):
        insert(lst, i)


if __name__ == '__main__':
    lst = [1, 6, 2, 7, 5]
    insert_sort(lst)
    print(lst)
```

第1个元素单独看做一个数列，它本身就是有序的，那么只需要执行insert(lst, 1)，就可以保证前两个数据变成有序的，然后执行insert(lst, 2)，此时，从索引0到索引1是有需的，只需要将

索引为2的数据插入到合适的位置就可以了。

6. 简单算法篇

6.1 打印杨辉三角

题目要求

给定一个正整数N,打印杨辉三角的前N行
杨辉三角形态如下

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

杨辉三角的每一行第一个和最后一个元素都是1
中间的元素,由上一行的两个元素相加得到,第N行的第index元素
是由第N-1 行的第index-1元素和第index相加得到的

思路分析

- 杨辉三角的每一行的第一个元素和最后也元素都是1
- 中间的元素可以由上一行的元素两两相加得到

第N行元素，可以由第N-1 行变化而来，这正是递归算法的精髓，把新的问题转化成老的问题，而老的问题转化成更老的问题，最终必然有一个老问题给出答案，然后整个逻辑链条上的问题都有了答案

代码示例

```
def print_yanghui(n):
    """
    递归算法打印杨辉三角
    :param n:
    :return:
    """
    if n == 1:
```

```
        print([1])
        return [1]
    elif n == 2:
        print_yanghui(1)
        print([1, 1])
        return [1, 1]  # 这里如果不返回,第三行就无法生成
    else:
        lst = [1]  # 第一个元素
        pre_lst = print_yanghui(n - 1)  # 得到上一行的元素
        # 根据第n -1 行的元素,生成第n行的中间元素
        for i in range(len(pre_lst) -1):
            lst.append(pre_lst[i] + pre_lst[i+1])
        lst.append(1)  # 最后一个元素
        print(lst)
        return lst

if __name__ == '__main__':
    print_yanghui(6)
```

6.2 计算三角形的周长和面积

题目要求

写一段程序，让用户输入三角形的三条边长，如果三条边长不能构成三角形，则提示用户重新输入

如果可以构成三角形，则计算周长和面积

思路分析

对于用户的输入，首先要约定格式，这里简单的约定为每个边长之间用空格间隔

在获得用户的输入以后，要对输入进行检查，有两点需要检查

（1） 检查是不是输入了三条边的边长，输入2个或者4个都是错误的

（2） 检查输入的内容是不是数值型，如果输入的是字母，那根本驴唇不对马嘴

以上两点是编程时要考虑的，经过上面的分析你应该有所体会，编程，并不是你掌握一门语言然后用它在计算机上做各种操作

编程，是对问题的思考，我这里约定让用户一次性输入三条边的边长，中间用空格隔开，你也可以让用户输入三次，也可以让用户输入一次但用别的字符做间隔，这些都是没有定论的，完全取决于你的思考

对于输入内容检查，你可能会以为python会自己完成，但其实不会，input获得的就是字符串，你必须理解什么是字符串，必须清楚的知道input的作用，这些都是最最基础的内容，如果你不掌握这些，那你就无法思考

根本不存在一种方法或者操作可以满足你的要求，可以解决实际的问题，编程就是分析问题然后你自己解决问题的过程

```
def get_edge(line):
    """
    根据用户的输入获得三条边
    用户输入三条边的长度,用空格隔开 类似于  3 4 5
    :param line:
    :return:
    """
    edge_lst = line.split(" ")
    # 如果输入条数不是3
    if len(edge_lst) != 3:
        return False, (0, 0, 0)
    try:
        # raw_input 获得用户的输入,得到的是字符串,这里把字符串转成float数值
        edge_lst = [float(item) for item in edge_lst]
    except:
        return False, (0, 0, 0)

    return True, (edge_lst[0], edge_lst[1], edge_lst[2])


def is_able_triangle(a, b, c):
    """
    判断能否构成三角形
    :param a:
    :param b:
    :param c:
    :return:
    """
    return (a + b > c) and (a + c > b) and (c + b > a)


def triangle_func():
    while True:
        line = input('输入三角形的三个边长,用空格隔开,退出请输入q:')
        if line == 'q':
            break
        input_correct, edges = get_edge(line)
        if not input_correct:
```

```
            print('输入错误')
            continue

        if not is_able_triangle(edges[0], edges[1], edges[2]):
            print('不能构成三角形')
            continue

        perimeter = sum(edges)
        half_perimeter = perimeter//2
        area = (half_perimeter*(half_perimeter-edges[0])*(half_perimeter-edges[1])*
(half_perimeter-edges[2])) ** 0.5

        print('周长: {perimeter}  面积:{area}'.format(perimeter=perimeter, area=are
a))

if __name__ == '__main__':
    triangle_func()
```

6.3 忽略大小比较字符串是否相等

题目要求

比较两个字符串，忽略大小写，比如字符串"abc"和字符串"ABC",在忽略大小写的情况下是相等的，实现函数

```
def is_same_ignore_case(str1, str2):
    """
    忽略大小写比较两个字符串是否相等
    :param str1:
    :param str2:
    :return:
    """
    pass
```

要求，不能使用字符串的lower方法和upper方法

思路分析

题目要求不能使用字符串的lower方法和upper方法，之所以这样要求，是希望你能从更基础的编程做起，培养更深入的思考。

要解这个练习题，你需要对ASCII码表有一定的了解，在ASCII码表中，大小写字母的十进制

编码相差32。

通过for循环，遍历str1,同时获取与之相对应索引上的字母，因为要忽略大小写，因此，将他们都转换成ASCII码表里的十进制数值，并且统一转成小写字母的十进制数值。

在进行正式比较前，严谨的判断逻辑应该是先判断传入的参数str1和str2是否有效，然后判断这两个字符串的长度是否相等，这些都是从程序的健壮性和效率上考虑的。

enumerate函数是一个非常方便且有用的函数，在for循环中，既能获得遍历的元素，又能获得该元素的索引。

示例代码

```
def get_ord_value(string):
    value = ord(string)
    if 65 <= value <= 90:
        value += 32

    return value


def is_same_ignore_case(str1, str2):
    """
    忽略大小写比较两个字符串是否相等
    :param str1:
    :param str2:
    :return:
    """
    if not isinstance(str1, str) or not isinstance(str2, str):
        return False

    if len(str1) != len(str2):
        return False

    # 统一转成小写
    for index, item in enumerate(str1):
        value1 = get_ord_value(item)
        value2 = get_ord_value(str2[index])

        if value1 != value2:
            return False

    return True
```

```
if __name__ == '__main__':
    print(is_same_ignore_case('ABC', 'abc'))
    print(is_same_ignore_case('ABC', 'abd'))
```

6.4 寻找数组最大值和最小值

题目要求

只遍历一遍，寻找出数组中的最大值和最小值，实现函数

```
def find_max_min(lst):
    pass
```

思路分析

如果只是要去你使用for循环找出列表里的最大值或者最小值，是一件非常简单的事情。题目要求实现函数find_max_min，找出列表的最大值和最小值，首先你要明白，python中，函数可以有多个返回值。

在for循环中，使用两个if条件语句，分别寻找最大值和最小值就可以了。

实例代码

```
def find_max_min(lst):
    if not lst or not isinstance(lst, list):
        return None, None

    max_value, min_value = lst[0], lst[0]
    for item in lst:
        if item > max_value:
            max_value = item
        if item < min_value:
            min_value = item

    return max_value, min_value


if __name__ == '__main__':
    lst = [43, 1, 12, 435, 456, 13]
    max_value, min_value = find_max_min(lst)
    print(max_value, min_value)
```

6.5 先递增后递减数组最大值

题目要求

一个数组先递增后递减，要求找到最大值，数组示例

```
lst = [1, 3, 6, 8, 5, 3, 2]
```

思路分析

既然是先递增，后递减，这样的数组至少应该有3个元素，才能符合这样的描述，而且，最大值一定不是首元素或者末尾元素。

从索引0开始遍历数组，如果下一个元素小于当前元素，那么当前这个元素一定是数组中最大的那个数据

示例代码

```
def find_max(lst):
    if not lst or not isinstance(lst, list):
        return None

    for index, item in enumerate(lst):
        if item > lst[index+1]:
            return item


if __name__ == '__main__':
    lst = [1, 3, 6, 8, 5, 3, 2]
    max_value = find_max(lst)
    print(max_value)
```

拓展

上面的算法并不是最高效的，鉴于其有序性，也可以使用二分法进行查找，对于初学者稍微有些困难，感兴趣的朋友可以自己去探索。

6.6 生成矩阵

题目要求

已知两个列表

```
lst_1 = [1, 2, 3, 4]
lst_2 = ['a', 'b', 'c', 'd']
```

请写算法，将两个列表交叉相乘，生成如下的矩阵

```
[['1a', '2a', '3a', '4a'],
 ['1b', '2b', '3b', '4b'],
 ['1c', '2c', '3c', '4c'],
 ['1d', '2d', '3d', '4d']]
```

思路分析

观察生成的矩阵，可以得出这样的结论，lst_1的长度决定了矩阵有多少列，lst_2的长度决定了生成的矩阵有多少行。

既然是交叉相乘，那么可以写两个for循环，嵌套遍历这两个列表，对lst_2的遍历放在外层，对lst_1的遍历放在内层。

示例代码

```
import pprint


lst_1 = [1, 2, 3, 4]
lst_2 = ['a', 'b', 'c', 'd']

lst = []
for item2 in lst_2:
    tmp = []
    for item1 in lst_1:
        tmp.append(str(item1) + item2)
    lst.append(tmp)

pprint.pprint(lst)
```

6.7 矩阵对角线元素和

题目要求

求一个3*3矩阵中对角线上元素之和
先模拟一个3*3矩阵

3 5 6

4 7 8

2 4 9

思路分析

在C语言里，这种数据要用二维数组来存储，在python里，没有二维数组这个专业用语，概念上，你可以理解为嵌套列表，其定义如下

```
lst = [
    [3,5,6],
    [4,7,8],
    [2,4,9]
]
```

lst中有3个元素，均是列表，lst[0]是一个列表，该列表里有3个元素。

题目要求计算对角线元素和，也就是lst[0][0] + lst[1][1] + lst[2][2],写一个for循环，用range(3)产生一个从0到2的序列，即可实现这三个元素的相加

实例代码

```
lst = [
    [3,5,6],
    [4,7,8],
    [2,4,9]
]
sum = 0
for i in range(3):
    sum += lst[i][i]

print sum
```

6.8 输出今天的信息

按照下面的格式，输出今天的时间信息

```
今天是2019年4月18日，星期四，今年的第108天，这一年29.59%的时间已流逝
```

思路分析

对日期的操作，使用datetime模块

```
today = datetime.datetime.now()
```

today存储了今天的日期信息，包括年月日，时分秒。

today.isoweekday() 返回的是数字1到7，对应周一到周日。

计算距离今年第一天的天数方法如下

```
days = int(today.strftime('%j'))
```

计算时间流逝的百分比，需要计算出今年一共有多少天，如果是闰年，是366天，本练习题并不复杂，考察你对datetime模块的熟练程度

示例代码

```
import datetime

out_put_str = "今天是{date_str}，{weekday}，今年的第{days}天，这一年{pass_ratio}%的时间已流逝"
year_days = 365   # 一年有365天

today = datetime.datetime.now()
date_str = '{year}年{month}月{day}日'.format(year=today.year, month=today.month, day=today.day)

year = today.year
# 判断是否为闰年，闰年的条件：  能被100乘除时，如果可以被400乘除，那么是闰年，不能被100乘除，能被4整除是闰年
b_runnian = False
if year % 100 == 0:
    if year % 400 == 0:
        b_runnian = True
elif year % 4 == 0:
    b_runnian = True

if b_runnian:
```

```
    year_days = 366


# 今年的第几天
days = int(today.strftime('%j'))



# 数字星期与中文星期的映射关系
week_map = {
    1: '星期一',
    2: '星期二',
    3: '星期三',
    4: '星期四',
    5: '星期五',
    6: '星期六',
    7: '星期日',
}

# 星期几
week_day = week_map[today.isoweekday()]
# 已经过去了多少
pass_ratio = round((days / year_days)*100, 2)
out_put = out_put_str.format(date_str=date_str, weekday=week_day, days=days, pass_r
atio=pass_ratio)

print(out_put)
```

6.9 可变对象与不可变对象考察

下面的题目，要求你在不执行程序的情况下说出程序的执行结果，然后，执行程序，将程序输出结果与自己的理解进行对比

6.9.1 不可变对象考察

```
lst = [1, 2, 3, 4, 5]
for item in lst:
    item = 0

print(lst)
```

上面的代码输出结果是

```
A. [1, 2, 3, 4, 5]
B. [0, 0, 0, 0, 0]
```

答案是 A， 在遍历过程中，item的数据类型是int,是不可变对象，item = 0仅仅是重新对item变量进行赋值，改变的是item变量的指向，并没有改变列表内容

6.9.2 可变对象考察

```
lst = [[1, 2, 3, 4], [5, 6, 7, 8]]

for item in lst:
    item.append(0)

print(lst)
```

请写出上面程序的输出结果
答案是

```
[[1, 2, 3, 4, 0], [5, 6, 7, 8, 0]]
```

遍历过程中，item的数据类型是列表，列表是可变对象，item.append(0) 这个操作，没有改变item变量的指向，而是修改了它所指向的列表里的内容

6.9.3 不可变对象考察

```
my_dic = {
    'class1': 90,
    'class2': 95,
    'class3': 100
}

for key, value in my_dic.items():
    value = 100

print(my_dic)
```

请输出上面程序的输出结果

答案是my_dic不发生任何变化，原理与6.9.1相同

6.9.4 可变对象考察

```
my_dic = {
    'class1': [1,2],
    'class2': [3,4],
    'class3': [5,6]
}

for key, value in my_dic.items():
    value.append(0)

print(my_dic)
```

请输出上面程序的输出结果

答案是

```
my_dic = {
    'class1': [1,2,0],
    'class2': [3,4,0],
    'class3': [5,6,0]
}
```

原理与6.9.2相同

6.10统计日期间隔

题目要求

需要编写一个函数get_day_diff(date_lst, target)， 入参示例如下

```
date_lst = [
                '2019-01-01', '2019-01-15',
                '2019-01-30', '2019-02-01',
                '2019-02-05', '2019-02-15',
                '2019-03-06', '2019-03-15',
                '2019-04-01', '2019-04-05',
                '2019-04-13', '2019-04-30',
                '2019-05-05', '2019-05-06'
                ]
target = '2019-05-08'
```

函数计算date_lst里的日期与target的间隔天数，然后统计这些天数信息，最后返回的结果示例如下

```
{'7_days': 2, '30_days': 4, '90_days': 9, '180_days': 14}
```

7_days 表示时间间隔小于7天的日期个数

思路分析

不可能直接用字符串计算日期的间隔，需要将这些字符串转成datetime类型，这样才能计算两个日期的间隔

最终的结果需要用字典来保存，因此函数里需要初始化一个字典

```
info = {
        '7_days': 0,
        '30_days': 0,
        '90_days': 0,
        '180_days':0
    }
```

用列表里的日期和target求间隔，然后做统计，如果间隔天数小于等于7天，则info['7_days'] += 1, 需要注意的地方是，一共有4个条件判断，而且这些条件判断之间不是互斥的关系，不能使用if else 这种逻辑判断，只需要4个if判断即可

示例代码

```
import datetime


def get_day_diff(date_lst, target):
    info = {
        '7_days': 0,
        '30_days': 0,
        '90_days': 0,
        '180_days':0
    }

    target_date = datetime.datetime.strptime(target, '%Y-%m-%d')
    for date_str in date_lst:
        date = datetime.datetime.strptime(date_str, '%Y-%m-%d')
        day_diff = (target_date - date).days
```

```
        if day_diff <= 180:
            info['180_days'] += 1

        if day_diff <= 90:
            info['90_days'] += 1

        if day_diff <= 30:
            info['30_days'] += 1

        if day_diff <= 7:
            info['7_days'] += 1

    return info


if __name__ == '__main__':
    date_lst = [
                '2019-01-01', '2019-01-15',
                '2019-01-30', '2019-02-01',
                '2019-02-05', '2019-02-15',
                '2019-03-06', '2019-03-15',
                '2019-04-01', '2019-04-05',
                '2019-04-13', '2019-04-30',
                '2019-05-05', '2019-05-06'
                ]

    info = get_day_diff(date_lst, '2019-05-08')
    print(info)
```

6.11 修改字典里的value

题目要求

有一个字典，保存的是学生各个编程语言的成绩，内容如下

```
data = {
    'python': '90',
    'c++': '95',
    'java': '90'
}
```

请编写函数 transfer_score，将value修改成int类型，最终字典内容变成

```
data = {
    'python': 90,
    'c++': 95,
    'java': 90
}
```

完善下面的代码

```
def transfer_score(score_dict):
    """
    在这里实现你的算法
    :param score_dict:
    :return:
    """
    pass


if __name__ == '__main__':
    data = {
        'python': '90',
        'c++': '95',
        'java': '90'
    }
    transfer_score(data)
    print(data)
```

思路分析

字典的结构非常简单，只需要遍历字典然后修改value类型即可，但如果你写成value = int(value) 是不会起到任何作用的，因为这样并没有修改真正的value，不要忘记了，想要修改字典里的value，必须通过key来修改，只能写成score_dict[key] = int(value)

示例代码

```
def transfer_score(score_dict):
    """
    在这里实现你的算法
    :param score_dict:
    :return:
    """
    for key, value in score_dict.items():
        score_dict[key] = int(value)
```

```
if __name__ == '__main__':
    data = {
        'python': '90',
        'c++': '95',
        'java': '90'
    }
    transfer_score(data)
    print(data)
```

6.12打印菱形

题目要求

实现函数print_diamond(count)，根据参数大小在屏幕上输出菱形

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

默认传入的参数一定是奇数，如果是7，则菱形最宽的地方有7个星号

思路分析

将菱形分为两部分，第一部分是从第一行到最宽的那一行，第二部分从最宽的那一行到最后一行

第一部分，星号的数量从1开始递增，每次增加两个，第二部分，星号的数量从count个开始递减，每次减少两个，与之对应的，第一部分，空格的数量是递减的。

count是多少，这个菱形就有多少行，使用for循环，让变量i从0变化到count-1，第一部分里，星号的数量和i之间的关系是2*i-1,空格的数量与i之间的关系是count//2-i，数学上，当i<=count//2时，都是第一部分图形。

进入第二部分后，空格与i之间的关系是i-count//2， 星号数量与i之间的关系(count-1)*2-

1。

本练习题，希望能纠正你对编程的错误认知和理解，编程不是你以为的发挥个人想象力进行创造，如同搭积木一般自由自在随意操作，编程需要的是缜密的思维，严谨的逻辑思考，你看到的例子都是简单几行代码就实现了惊艳的功能，因此被忽悠的学习了python，但那些惊艳的例子不是编程的全部，甚至除了勾引你入门意外，什么都不会交给你。

本题展示出的，才是编程的本质，用程序语言表达出你对问题的思考，逻辑推理，解决一个实际问题所需要的，是扎实的算法功底，同学，努力吧。

示例代码

```
def print_diamond(count):
    for i in range(count):
        if i <= count//2:
            print(" "*(count//2-i)  + "*"*(2*i + 1))
        else:
            print(" "*(i - count//2)  + "*"*((count-i)*2-1))

print_diamond(5)
```

6.13 打印九九乘法表

题目要求

在屏幕上输出九九乘法表

```
1*1 = 1
1*2 = 2  2*2 = 4
1*3 = 3  2*3 = 6  3*3 = 9
1*4 = 4  2*4 = 8  3*4 = 12  4*4 = 16
1*5 = 5  2*5 = 10  3*5 = 15  4*5 = 20  5*5 = 25
1*6 = 6  2*6 = 12  3*6 = 18  4*6 = 24  5*6 = 30  6*6 = 36
1*7 = 7  2*7 = 14  3*7 = 21  4*7 = 28  5*7 = 35  6*7 = 42  7*7 = 49
1*8 = 8  2*8 = 16  3*8 = 24  4*8 = 32  5*8 = 40  6*8 = 48  7*8 = 56  8*8 = 64
1*9 = 9  2*9 = 18  3*9 = 27  4*9 = 36  5*9 = 45  6*9 = 54  7*9 = 63  8*9 = 72  9*9
= 81
```

思路分析

题目乍看起来挺难的，但其实非常简单，前提是你知道如何分析。

九九乘法表一共有9行，我们先考虑如何输出一行，一行解决了，使用一个for循环，就可以把所有行都输出了，定义函数print_line(line_number)，该函数输出第line_number行。

当line_number等于9的时候，研究一下如何输出第9行。第9行有9个式子，从1到9分别乘以9，看见没，这不就是一个for循环么，使用一个for循环，让变量i从1变化到9，每次都与9相乘，并将结果组装成一个i*9 = xx 的式子，把所有的式子连接在一起，就是第9行的内容

解决了指定行的输出后，只需要一个简单的for循环，从1到9，分别去调用这个函数不就将整个乘法表打印出来了么?

示例代码

```
def print_line(line_number):
    lst = []
    for i in range(1, line_number+1):
        part = "{index}*{number} = {res}".format(index=i, number=line_number, res=i
*line_number)
        lst.append(part)

    print("  ".join(lst))


def print_table():
    for i in range(1, 10):
        print_line(i)


print_table()
```

6.14 统计字符数量

题目要求

使用input函数接收用户的输入，统计出其中英文字母、空格、数字和其它字符的个数，例如用户输入字符串"sdfijer384323js",程序最终输出信息

```
{'s': 2, 'd': 1, 'f': 1, 'i': 1, 'j': 2, 'e': 1, 'r': 1, '3': 3, '8': 1, '4': 1, '2
': 1}
```

思路分析

input函数接收用户的输入，得到的是字符串，遍历字符串，统计字符串中每一个字符出现的次数，这种信息自然是用字典来存储，其他容器类型数据列表，元组，集合，都不适合存储这种key-value形式的数据

示例代码

```
string = input("请输入一个字符串:")
info = {}
for item in string:
    info.setdefault(item, 0)
    info[item] += 1

print(info)
```

6.15 水仙花数

输出所有的水仙花数，所谓水仙花数是指一个三位数，各个位上的数的立方相加在一起等于这个三位数，比如153，1的3次方 + 5的三次方 + 3的三次方 等于153

思路分析

水仙花数是一个3位数，数值范围是从100到999之间，写一个for循环，从100到999进行遍历，逐个判断该数是否为水仙花数即可。

对于一个三位数，获得这个数的每一位也并不难，以153为例

```
a = 153
while a > 0:
    print(a % 10)
    a //= 10
```

对数字的操作，永远离不开取模运算和整除运算

示例代码

```
for i in range(100, 1000):
    res = 0
    value = i
    while value > 0:
        res += (value%10)**3
        value //= 10
```

```
    if res == i:
        print(res)
```

6.16 完全平方数

完全平方数，就是可以表示为某个整数的平方的数，例如9，是3的平方，16是4的平方，9和16都是完全平方数，请打印10000以内的完全平方数

思路分析

两个思路，一个思路是从1到10000进行遍历，对每一个数值进行判断，判断其是否为某个整数的平方。

第二个思路，从1到10000进行遍历，计算每一个数值的平方，如果这个平方小于10000，那么这个数值的平方就是完全平方数。

显然，第二个方法更容易一些，毕竟开根号这种事情，不能保证开出来的一定是整数。

示例代码

```
i = 1
value = 1
while value < 10000:
    print(i, value)
    i += 1
    value = i**2
```

6.17求三位数组合

```
lst = [3, 6, 2, 7]
```

这四个数字能组成多少个互不相同且无重复数字的三位数？比如362算一个，326算一个，请逐个输出他们

思路分析

从4个数字里挑出来，组成一个3位数，就算法而言，最方便的做法是用一个3层嵌套循环，分别从lst取数，取出来的数值组成一个3位数，题目要求无重复数字，这就要求，取出来的3个

数字互不相等

示例代码

```
lst = [3, 6, 2, 7]

for a in lst:
    for b in lst:
        for c in lst:
            if a != b and b != c and a != c:
                print(a*100 + b*10 + c)
```

如何判断3个数值互不相等，还有一个更简单的办法

```
if a not in (b, c) and b != c:
```

6.18 翻转列表

题目要求

```
lst = [1,2,3,4,5]
翻转后
lst = [5,4,3,2,1]
```

思路分析

如果是用列表自带的功能，翻转列表是非常容易的事情

```
lst = lst[::-1]
```

现在，需要你自己来实现翻转过程，思路很简单，从头开始遍历列表，但只遍历到一半，左右两边对位交换数据即可

示例代码

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
length = len(lst)

for i in range(length//2):
    tmp = lst[i]
    lst[i] = lst[length - 1 - i]
    lst[length - 1 - i] = tmp

print(lst)
```

6.19 列表偏移

题目要求

lst = [1,2,3,4,5]，列表向右偏移两位后，变成lst = [5,4,1,2,3]

思路分析

先说简易思路，以向右偏移2位做例子，直接对列表进行切片操作，然后再讲切片后的列表连接在一起

```
lst = [1,2,3,4,5]

lst = lst[len(lst)-2:] + lst[:len(lst)-2]
print(lst)
```

如果是初学者这样做，是完全可以接受的，但如果想更上一层楼，则需要更加合理的解决方法

首先，偏移的位数可能会超过列表的长度，比如向右偏移47位，此时，就要考虑用47%5 = 2，向右偏移47位等价于向右偏移2位

直接对列表进行切片操作，终究还是用了列表的原生功能，既然是练习题，且偏向于算法层面，那么我们应该自己来实现偏移，不生成新的列表，只在原列表上动手脚

不论偏移多少位，在实现时，每一次，我们只偏移1位，先实现偏移1位的情况。

偏移1位，只需要将列表最后一位记录下来，然后其他所有位置上的数值向右移动一位，最后将之前记录下来的列表最后一位放置在列表的开头即可。

实现了偏移1位，剩下的仅仅是写一个for循环，就可以偏移指定的位数

示例代码

```
lst = [1,2,3,4,5]

count = int(input("请输入偏移位数"))
count = count % len(lst)

for i in range(count):
    tmp = lst[-1]
    for j in range(len(lst)-1, 0, -1):
        lst[j] = lst[j-1]

    lst[0] = tmp

print(lst)
```

6.20 比较三个数的大小

使用input函数接收用户的输入，用户输入三个整数，中间用空格分隔，程序判断这三个数的大小关系，最后用print函数输出这个三个数，从大到小

```
输入：  43 22 88
输出：  88 43 22
```

思路分析

其实，真没啥好分析的

- input函数返回的是字符串
- 我们需要的是int类型的数据，因此用split函数对字符串进行分割
- 分割后，三个部分都转换成int类型的数据
- 比较大小就是一个简单的事情了，纯粹是数学问题，小学生都会，只是你要把各种逻辑分支都考虑全，这种题目考察锻炼的是你的思维能力

示例代码

```
string = input("请输入三个整数,中间用空格分隔:")
lst = string.split()
a, b, c = int(lst[0]), int(lst[1]), int(lst[2])

if a >= b:
    if c >= a:
        print(c, a, b)
```

```
    elif c >= b:
        print(a, c, b)
    else:
        print(a, b, c)
else:
    # b > a
    if c >= b:
        print(c, b, a)
    elif c >= a:
        print(b, c, a)
    else:
        print(b, a, c)
```

6.21 求学生最高分数科目和分数

题目要求

实现下面的函数

```
def get_max_socre(score_dic):
    """
    返回学生考试成绩的最高分的科目和分数
    :param score_dic:
    :return:
    """
    pass
```

传入的socre_dic 内容类似下面的数据

```
dic = {
    '语文': 90,
    '数学': 97,
    '英语': 98
}
```

程序最终输出: 英语 98

思路分析

万变不离其中，遍历是万能的，遍历传入的字典，用一个变量保存最高分数，用另一个变量保存科目，遇到更高的分数，则修改这两个变量，就是这么的简单

示例代码

```
def get_max_socre(score_dic):
    """
    返回学生考试成绩的最高分的科目和分数
    :param score_dic:
    :return:
    """
    max_score = 0
    max_course = ''

    for key, value in score_dic.items():
        if value > max_score:
            max_score = value
            max_course = key

    print(max_course, max_score)


if __name__ == '__main__':
    dic = {
        '语文': 90,
        '数学': 97,
        '英语': 98
    }

    get_max_socre(dic)
```

6.22 水果

小明喜欢葡萄，香蕉，苹果，小红喜欢香蕉，桃子，草莓， 请写程序分析

1. 他们共同喜欢的水果是什么?
2. 他们两个喜欢的水果加在一起都有什么
3. 什么水果是小明喜欢的却不是小红喜欢的?
4. 什么水果是小红喜欢的却不是小明喜欢的

思路分析

交集，差集，并集，这些数据操作使用集合最为合适

示例代码

```
lst_1 = ['葡萄', '香蕉', '苹果']
lst_2 = ['香蕉', '桃子', '草莓']

set_1 = set(lst_1)
set_2 = set(lst_2)

print(set_1.intersection(set_2))
print(set_1.union(set_2))
print(set_1.difference(set_2))
print(set_2.difference(set_1))
```

7 中等难度算法练习题

7.1 验证回文串

题目要求

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写，例如“123A man, a plan, a canal: Panama321”

严格的讲，这个字符串并不是一个回文，但是如果只考虑字母和数字并且忽略大小写，那么它确实是一个回文

思路分析

这道题目非常简单，需要掌握字符串的两个基本方法

- isalpha() 判断字符串是否是字母
- isdigit() 判断字符串是否是数字

逐一遍历字符串，判断每一个字符是否符合字母和数字的要求，把符合要求的字符串放入到一个list中，最后用列表的join方法将列表中的字符串拼接成一个新的字符串，题目还要求忽略大小写，新的字符串转成小写即可。

经过一番处理，从原始字符串得到了一个全是小写的只包含字母和数字的字符串，那么剩下的事情就变得非常简单了。

示例代码

```
# coding=utf-8
```

```python
def isPalindrome(string):
    # 把字符串中的字母和数字挑选出来
    str_lst= []
    for item in string:
        if item.isalpha() or item.isdigit():
            str_lst.append(item)

    # 组成新的字符串,并转成小写
    new_string = "".join(str_lst).lower()
    for i in range(len(new_string)/2):
        if new_string[i] != new_string[len(new_string)-1-i]:
            return False
    return True


if __name__ == '__main__':
    string = "123A man, a plan, a canal: Panama321"
    print(isPalindrome(string))
```

7.2 翻转字符串里的单词

题目要求

给定一个字符串，逐个翻转字符串中的每个单
示例:

```
输入: " the sky is   blue",
输出: "blue   is sky the "
```

翻转后，空格不能减少，单词之间的空格数量不能发生变化

思路分析

如果只是单纯的翻转字符串，就过于简单了，因此要求翻转每一个单词，单词还是原来的样子，但是单词所在的位置却发生了翻转，第一个单词变成了倒数第一个单词。

可以先将整个字符串都翻转，这样单词的位置完成了翻转，结果如下：

```
"eulb   si yks eht "
```

但这个时候，每一个单词的顺序还是颠倒的，因此需要对每一个单词进行一次翻转。

字符串是不可变对象，不能直接在字符串上进行翻转，要借助列表（list）进行翻转，翻转后在使用join方法将列表里的元素拼成字符串

示例代码

```
# coding=utf-8


def reverse_word(string):
    # 先将整个句子翻转
    word_lst = list(string)
    for i in range(len(word_lst)/2):
        word_lst[i], word_lst[len(word_lst)-1-i] = word_lst[len(word_lst)-1-i], wor
d_lst[i]

    print("".join(word_lst))
    # 翻转里面的单词
    start_index = None
    end_index = None
    b_word = False
    for index, item in enumerate(word_lst):
        if item.isalpha():
            if not b_word:
                start_index = index     # 单词开始的位置
                b_word = True
        else:
            if b_word:
                end_index = index -1    # 单词结束的位置
                b_word = False
                # 已经知道单词开始和结束的位置,翻转这个单词
                reverse_single_word(word_lst, start_index, end_index)

    return "".join(word_lst)


def reverse_single_word(lst, start_index, end_index):
    """
    将lst 中 start_index到end_index 这个区域翻转
    :param lst:
    :param start_index:
    :param end_index:
    :return:
    """
```

```
    while start_index < end_index:
        lst[start_index], lst[end_index] = lst[end_index], lst[start_index]
        start_index += 1
        end_index -= 1


if __name__ == '__main__':
    print(reverse_word(" the sky is   blue"))
```

7.3 寻找峰值

题目要求

峰值元素是指其值大于左右相邻值的元素，给定一个序列，请返回所有的峰值元素及其索引

示例1：

```
输入: nums = [1,2,3,1]
输出: [(2, 3)]
```

以列表形式输出，元素为tuple，tuple[0]表示索引，tuple[1]表示元素

思路分析

遍历序列，对于每一个元素，都要和自己左右两侧的相邻值进行比较，如果大于两侧相邻值，那么这个元素就是峰值元素。

需要注意的是序列的两端元素，他们都只有一侧有相邻元素，因此在逻辑处理上要考虑这种边界情况。

示例代码

```
def find_peak(lst):
    if len(lst) <= 2:
            return -1
    peak_lst = []
    for index, item in enumerate(lst):
        big_than_left = False       # 是否比左侧大
        big_than_right = False      # 是否比右侧大
        # 考虑边界情况
        if index == 0:
            big_than_left = True
```

```
            big_than_right = item > lst[index+1]
        elif index== len(lst) - 1:
            big_than_right = True
            big_than_left = item > lst[index-1]
        else:
            big_than_left = item > lst[index-1]
            big_than_right = item > lst[index+1]

        if big_than_left and big_than_right:
            peak_lst.append((index, item))

    return peak_lst


if __name__ == '__main__':
    lst = [1, 2, 1, 3, 5, 6, 4, 8]
    print(find_peak(lst))
```

7.4 字符串转整数

题目要求

实现函数atoi，将字符串转为整数，具体要求如下：

1. 字符串第一个非空字符如果不是数字，字符串无效，返回0
2. 字符串第一个非空字符如果是 + 或者 −，则表示正负，且该字符后面的第一个非空字符必须是数字，否则视为无效字符串，返回0
3. 遇到数字，将其与之后连续的数字字符组合起来形成整数
4. 有效数字范围是$[-2^{31}, 2^{31} - 1]$,数值超过可表示的范围，则返回$2^{31} - 1$ 或-2^{31} 。
   示例

```
1、 "42" 转成 42
2、 "   -  42 43" 转成 -4243
3、 "4193 word" 转成4193
4、 "word 4193" 转成0
5、 "-91283472332" 转成-91283472332,  比-2^31还小，返回-2^31
```

思路分析

基本思路是遍历字符串，遍历过程中，关键点是找到第一个非空字符，这个非空字符决定了接下来程序的走向。

如果第一个非空字符是+或者-，就决定了数值的正负，后面的工作是提取数值

如果第一个非空字符是数字，事情变得简单了，直接提取数值。

如果第一个非空字符既不是数字，也不是+或-，就是无效字符串，返回0即可。

提取数值的过程，用一个列表保存单个数字，直到遇到一个既不是数字也不是空格的字符或者遇到字符串末尾，将列表里的数值连接起来并用int转成数值。

程序的最后，要和最大值和小值比较。

示例代码

```
# coding=utf-8
MAX_INT = 2**31-1
MIN_INT = -2**31


def atoi(string):
    """
    将字符串转成int
    :param string:
    :return:
    """
    if not isinstance(string, basestring):
        return 0
    if string == u'':
        return 0

    tag = 1    # 标识正负，赋值为1表示为正
    value = 0
    for index, item in enumerate(string):
        # 空字符不处理
        if item == u' ':
            continue
        elif item in ('+', '-'):
            if item == '-':
                tag = -1
            value = get_int(string, index)
            break
        elif item.isdigit():
            value = get_int(string, index)
            break
        else:
            return 0
```

```
    value = tag * value
    if value > MAX_INT:
        value = MAX_INT
    if value < MIN_INT:
        value = MIN_INT

    return value


def get_int(string, start_index):
    """
    提取数值
    :param string: 字符串
    :param start_index: 数值开始部分
    :return:
    """
    lst = []
    for index in range(start_index, len(string)):
        if string[index] == u' ':       # 空字符不用处理
            continue
        if string[index].isdigit():     # 遇到不是数字时结束
            lst.append(string[index])

    return int(''.join(lst))


if __name__ == '__main__':
    print atoi("42")
    print atoi("   -  42 43 ")
    print atoi("4193 4 word")
    print atoi("word 4193")
    print atoi("-91283472332")
```

7.5 判断数组是山脉数组

如果一个数组k符合下面两个属性，则称之为山脉数组

- 数组的长度大于等于3
- 存在i，i >0 且 i < len(k)−1， 使得k[0] < k[1]< ... < k[i−1] < k[i] > k[i+1] ... >k[len(k)−1]， 这个i就是顶峰索引。

现在，给定一个山脉数组，求顶峰索引。

```
输入: [1, 3, 4, 5, 3]
输出: True

输入: [1, 2, 4, 6, 4, 5]
输出: False
```

思路分析

山脉数组的特点是在到达山脉峰顶前，每个元素都比右侧的元素小，过了峰顶之后，每个元素都比右侧元素大。

对数组k进行遍历，当k[index] < k[index+1] 不成立时，说明可能到达峰顶了，记录当前这个索引index, 接下来要判断index的值，如果index等于0，说明k[0]<k[1] 不成立，显然不是山脉数组，如果index 等于len(k)–1， 说明倒数第2个数元素小于倒数第1个元素，显然也不是山脉数组。

如果index的值符合要求,也只能说明在index之前，是一个爬坡的过程，过了index之后，需要再用一个循环继续遍历，如果lst[index]>lst[index+1] 不成立，则不是山脉数组，前半段，后半段的判断简单一些，不涉及到峰顶索引位置的判断。

示例代码

```
# coding=utf-8

def is_mountain_lst(lst):
    # 如果长度小于3,不符合定义
    if len(lst) < 3:
        return False

    # 第一个循环找到发生转折的地方
    index = 0
    while index < len(lst)-1:
        if lst[index] < lst[index+1]:
            index += 1
        else:
            # 当前元素比右侧元素小,说明到达峰顶了,停止循环
            break

    # 如果index == 0,说明lst[0] < lst[1] 不成立,显然不是山脉数组
    # 如果index == len(lst) -1, 说明倒数第2个数小于倒数第一个数,显然也不是山脉数组
    if index == 0 or index== len(lst) -1:
        return False
```

```
    # 接下来要判断从index 开始到列表末尾,是不是都满足lst[index] > lst[index+1]
    while index < len(lst)-1:
        if lst[index] > lst[index+1]:
            index += 1
        else:
            return False

    return True

if __name__ == '__main__':
    print is_mountain_lst([1, 2])
    print is_mountain_lst([1, 2, 3])
    print is_mountain_lst([1, 3, 4, 5, 3])
    print is_mountain_lst([1, 2, 4, 6, 4, 5])
```

7.6 二进制中为1的位数

题目要求

给定一个整数，请计算二进制中为1的位数

```
输入: 13
输出: 3
解释: 13的二进制表示是 1101，位为1的数量是3
```

思路分析

如果一个数是奇数，那么它的二进制的最后一位一定是1，道理很简单，其他的位都表示2^n 只有最后一位表示2^0 。我们可以利用最后一位是否为1来统计为1的位数，这就需要最后一位是变化的，还好，我们可以用位运算符 >> (右移位运算符)

13 的 二进制表示是 1101 ，13>>1 就表示二进制的每一位都向右移动一位，移动后为 110，最右边的1舍弃。如果二进制最后一位是1，那么一定是奇数。

示例代码

```
# coding=utf-8


def count_one_bits(value):
    count = 0
```

```
    # 当value等于0时,二进制的每一位都是0
    while value:
        if value % 2 == 1:
            count += 1
        value = value >> 1
    return count


if __name__ == '__main__':
    print(count_one_bits(6))
    print(count_one_bits(13))
```

小小的升级

上面的分析中，利用奇偶数来判断最后一位是否为1，除此以外，还可以利用位运算符 &（按位与）来判断最后一位是否为1。

13 的二进制是 1101 ，1的二进制是1，13&1 等价于1101 & 0001， 相同位进行与运算，得到结果为0001。

示例代码

```
# coding=utf-8


def count_one_bits(value):
    count = 0
    # 当value等于0时,二进制的每一位都是0
    while value:
        if value & 1 == 1:
            count += 1
        value = value >> 1
    return count


if __name__ == '__main__':
    print(count_one_bits(6))
    print(count_one_bits(13))
```

7.7 寻找缺失数字

题目要求

有一组数字，从1到n，中减少了一个数，顺序也被打乱，放在一个n-1的数组里，请找出丢失的数字。

```
输入：  5 3 1 2 0
输出: 4
```

思路分析

这道题目虽小，却有多种解法。

解法1 利用递增序列求和

把4放回序列中，那么整个序列的和就是(n*(n+1))/2 ,n = 5, 结果为 15，现在序列里没有4，5+3+1+2+0 = 11, 15-11恰好就是缺失的数字
示例代码：

```
def find_miss_number_1(lst):
    n = len(lst)
    sum_lst = n*(n+1)/2
    sum_lst_ex = sum(lst)
    miss_number = sum_lst - sum_lst_ex
    return miss_number


if __name__ == '__main__':
    lst = [5, 3, 1, 2, 0]
    print(find_miss_number_1(lst))
```

解法2 利用索引

示例中的序列有5个数字，算上丢失的有6个数字，创建一个长度为6的标识列表，元素都初始化为0，遍历序列，找到每个值在标识列表里的对应位置，修改元素值为1，哪个位置没有被修改为1，那个位置的索引就是丢失的值

```
def find_miss_number_2(lst):
    tag_lst = [0 for i in range(len(lst) + 1)]
    for item in lst:
        tag_lst[item] = 1

    for index, item in enumerate(tag_lst):
        if item == 0:
```

```
            return index

if __name__ == '__main__':
    lst = [5, 3, 1, 2, 0]
    print(find_miss_number_2(lst))
```

解法3 利用亦或运算

0 与任意一个数做亦或操作还是自身，一个数与自己做亦或操作则等于0。先不考虑缺失了哪个数字，用一个变量res 存储 0^1^2...^n，然后再用res逐个与序列里的值进行亦或操作，这样，没有缺失的值会与自己进行亦或操作得到0，而缺失的那个数字则最终被留存下来

```
def find_miss_number_3(lst):
    res = 0
    for i in range(1, len(lst)+1):
        res ^= i

    for item in lst:
        res ^= item

    return res


if __name__ == '__main__':
    lst = [5, 3, 1, 2, 0]
    print(find_miss_number_3(lst))
```

7.8 二进制求和

题目要求

计算两个二进制字符串的和

```
输入: 111  1110
输出: 10101
```

思路分析

参与计算的二进制字符串长度可以不同，这样为计算带来麻烦，所以，首先要补齐那个较短的字符串。如果较短字符串长度比长字符串小3，就在较短字符串前面补3个0.

计算的过程，可以模拟手动计算的过程，从右向左逐位计算。反向遍历字符串，相同位上都是1，相加后，这一位就变成了0，而且还要向前进位，下一位的计算要把进位的数值考虑进来，计算右数第一位时，进位设置为0。

示例代码

```
def binary_add(str_1, str_2):
    # 先补齐
    str_1_length = len(str_1)
    str_2_length = len(str_2)
    if str_1_length < str_2_length:
        str_1 = "0"*(str_2_length - str_1_length) + str_1
    else:
        str_2 = "0"*(str_1_length - str_2_length) + str_2

    # 进行计算
    index = len(str_1) - 1
    pre_num = 0          # 记录进位
    res_lst = []         # 记录结果

    # 方向遍历
    while index >= 0:
        item_1 = int(str_1[index])
        item_2 = int(str_2[index])
        item_sum = item_1 + item_2 + pre_num
        pre_num = item_sum // 2
        curr_num = item_sum % 2

        # 新的计算结果插入到结果的第一位
        res_lst.insert(0, str(curr_num))
        index -= 1

    if pre_num == 1:
        res_lst.insert(0, '1')

    return ''.join(res_lst)


if __name__ == '__main__':
    print(binary_add("111", '1110'))
    print(binary_add("11", '1'))
    print(binary_add("101", '1001'))
```

7.9 第一个只出现一次的字符

题目要求

一个只包含小写字母的字符串，请找出第一个只出现一次的字符，并返回索引，如果这样的字符不存在返回-1，不允许使用字典

```
输入：  abcacd
输出：  1
解释：  出现一次的字符是 b 和 d，b是第一个出现的
```

思路分析

必须遍历字符串，统计每个字符出现的次数，然后再遍历一遍字符串，如果某个字符出现的次数是1，则返回这个字符的索引。

题目要求不允许使用字典，因此，需要换一个数据类型来存储字符串出现的次数，这里可以使用列表，创建一个长度为26的列表，元素初始化为0。小写字母一共有26个，字符a 的ascii码10进制是97，字符z的ascii码10进制是122，遍历过程中，先将字符转成ascii码十进制数值并减去97，就得到了在列表中的索引，列表索引位置元素加1，这样，就解决了字符出现次数的记录问题。

再次遍历字符串时，还是对字符进行转换，转换后的数值作为索引找到列表中的值，如果值为1，那么这个字符就是出现次数为1的字符。

示例代码

```python
# coding=utf-8


def find_first_single_char(string):
    """
    寻找字符串中第一个只出现一次的字符
    :param string:
    :return:
    """
    # 记录每个字符出现的次数
    lst = [0 for i in range(26)]
    for item in string:
        lst[ord(item) - 97] +=1
```

```
    for index, item in enumerate(string):
        if lst[ord(item) - 97] == 1:
            return index

    return -1


if __name__ == '__main__':
    print(find_first_single_char('abcacd'))
```

7.10 字符串相乘

题目要求

有两个字符串，str1,str2，内容为数值，数值均大于0请编写算法计算两个字符串相乘的结果，不可以使用大数据类型，不可以把字符串直接转成整数来处理。

```
输入: str1='3'  str2='5'
输出: '15'

输入: str1='32'  str2='15'
输出: '480'

输入: str1='323434223434234242343'
     str2='15492828424295835983529'
输出:
```

思路分析

题目要求的很明确，不可以直接把字符串转成整数然后相乘，所以int(str1)*int(str2)是不被允许的。

不止如此，第三个示例里，数值已经非常大，已经超出了2^{64} ，在计算过程中，中间结果也非常的大，在大多数编程语言里，你找不到合适的数据类型来存储这样的数据，因此，只能用字符串来保存。

先来看第一个示例， 3 乘 5， 简单的乘法，在看第二个示例， 32 乘 15 ，小学的时候都学过如何计算乘法，32*5 + 32*10 =480。需要编写一个算法，来实现这个计算过程，但是要注意，32*5的结果必须保存成为字符串"160",为什么不能保存成数值型数据160呢？就32*15来说，保存成数值是没有问题的，可对于第三个示例，在计算中间结果时，得到的结果太大了，根本无法用数值型变量保存，因此必须保存成字符串。

最简单的乘法

我们需要先实现一个简单的字符串乘法，函数定义为

```
def single_str_multi(str1, single):
    """
    single是一个小于10的数值，例如 323354 * 4
    :param str1: 基础数值
    :param single: 单个数值
    :return:
    """
    pass
```

如果总是可以保证single是一个小于10的数值，那么计算起来就简单容易的多了，计算时，反向遍历str1，每个数值分别与single相乘，计算进位和当前位置留下的值，并存入一个列表中，遍历结束后，要注意检查最后一次进位的值，算法实现如下

```
def single_str_multi(str1, single):
    """
    single是一个小于10的数值，例如 323354 * 4
    :param str1: 基础数值
    :param single: 单个数值
    :return:
    """
    pre_sum = 0             # 进位
    single = int(single)
    res_lst = []
    for item in reversed(str1):
        # 每次计算都要考虑上一次计算结果的进位
        item_num = int(item) * single + pre_sum
        pre_sum = item_num / 10     # 计算进位
        curr_num = item_num % 10    # 计算当前位置的值
        res_lst.insert(0, str(curr_num))

    if pre_sum > 0:
        res_lst.insert(0, str(pre_sum))

    return ''.join(res_lst)
```

有了single_str_multi函数做基础，就可以实现最终的字符串相乘算法了，定义函数

```
def str_multi(str1, str2):
    pass
```

这一次，反向遍历str2，每次遍历都得到一个单一数值single，这样恰好可以调用 single_str_multi 函数，但是需要注意的是每次得到的结果都要根据single的位置补0，如果single是str2的百位，那么计算结果就要乘100。

字符串相加

每次调用single_str_multi函数，得到的都是中间结果，这些结果必须加在一起才能得到乘法的结果，因此，我们还需要一个计算字符串加法的函数，前面的计算二进制加法的练习题已经有过讲解，代码稍作修改即可

```
def str_sum(str1, str2):
    """
    计算两个字符串的加法
    :param str1:
    :param str2:
    :return:
    """
    # 先补齐
    str_1_length = len(str1)
    str_2_length = len(str2)
    if str_1_length < str_2_length:
        str1 = "0"*(str_2_length - str_1_length) + str1
    else:
        str2 = "0"*(str_1_length - str_2_length) + str2

    # 进行计算
    index = len(str1) - 1
    pre_num = 0          # 记录进位
    res_lst = []         # 记录结果

    # 反向遍历
    while index >= 0:
        item_1 = int(str1[index])
        item_2 = int(str2[index])
        item_sum = item_1 + item_2 + pre_num
        pre_num = item_sum/10
        curr_num = item_sum % 10

        # 新的计算结果插入到结果的第一位
        res_lst.insert(0, str(curr_num))
        index -= 1

    if pre_num == 1:
```

```
        res_lst.insert(0, '1')

    return ''.join(res_lst)
```

字符串相乘

万事具备，之前东风，最后来实现str_multi函数

```
def str_multi(str1, str2):
    """
    字符串相乘
    :param str1:
    :param str2:
    :return:
    """
    res = '0'
    for index, item in enumerate(reversed(str2)):
        if item == '0':   # 为0时不用计算
            continue
        # 一定要补0
        single_res = single_str_multi(str1, item) + '0'*index
        # 每次相乘结果要相加
        res = str_sum(res, single_res)
    return res
```

全部代码

```
def str_sum(str1, str2):
    """
    计算两个字符串的加法
    :param str1:
    :param str2:
    :return:
    """
    # 先补齐
    str_1_length = len(str1)
    str_2_length = len(str2)
    if str_1_length < str_2_length:
        str1 = "0"*(str_2_length - str_1_length) + str1
    else:
        str2 = "0"*(str_1_length - str_2_length) + str2

    # 进行计算
```

```
    index = len(str1) - 1
    pre_num = 0         # 记录进位
    res_lst = []        # 记录结果

    # 方向遍历
    while index >= 0:
        item_1 = int(str1[index])
        item_2 = int(str2[index])
        item_sum = item_1 + item_2 + pre_num
        pre_num = item_sum//10
        curr_num = item_sum % 10

        # 新的计算结果插入到结果的第一位
        res_lst.insert(0, str(curr_num))
        index -= 1

    if pre_num == 1:
        res_lst.insert(0, '1')

    return ''.join(res_lst)


def single_str_multi(str1, single):
    """
    single是一个小于10的数值, 例如 323354 * 4
    :param str1: 基础数值
    :param single: 单个数值
    :return:
    """
    pre_sum = 0            # 进位
    single = int(single)
    res_lst = []
    for item in reversed(str1):
        # 每次计算都要考虑上一次计算结果的进位
        item_num = int(item) * single + pre_sum
        pre_sum = item_num // 10     # 计算进位
        curr_num = item_num % 10    # 计算当前位置的值
        res_lst.insert(0, str(curr_num))

    if pre_sum > 0:
        res_lst.insert(0, str(pre_sum))

    return ''.join(res_lst)


def str_multi(str1, str2):
```

```
    """
    字符串相乘
    :param str1:
    :param str2:
    :return:
    """
    res = '0'
    for index, item in enumerate(reversed(str2)):
        if item == '0':   # 为0时不用计算
            continue
        # 一定要补0
        single_res = single_str_multi(str1, item)
        if single_res != '0':
            single_res += '0'*index
        # 每次相乘结果要相加
        res = str_sum(res, single_res)
    return res


if __name__ == '__main__':
    print(str_multi('323434223434234242343', '15492828424295835983529'))
```

7.11 整数翻转

题目要求

```
def reverse_int(number):
```

已知函数reverse_int，参数number是正整数，实现函数，将number翻转并返回,不要使用str函数将数值转成字符串
示例

```
输入: 12345
输出: 54321
```

思路分析

题目要求不能使用str函数将数值转成字符串，因为这样做实在是太简单了，下面是这样处理的示例代码

```
def reverse_int(number):
    str_number = str(number)
    lst = list(str_number)
    lst = lst[::-1]
    return int("".join(lst))

print(reverse_int(12345))
```

1. 将数值转成字符串
2. 将字符串转成list
3. 翻转list，注意翻转方法lst[::-1]
4. 使用join方法连接列表里的字符串，并最终转成int类型数据返回

既然不允许这样做，那么，我们就考虑别的方法
对于整数的操作，你一定要考虑求模运算和除法运算

- 对10求模可以得到一个整数的个位数
- 一个数除以10，可以去掉个位数

题目要求翻转，那么，我们可以先对10求模，就得到了个位数，而这个个位数就是翻转后的最高位，然后在除10，就去除了个位数。

讲过上面两步操作，我们得到了两个数，一个是个位数，一个是去除个位数后剩余的部分，对于这个剩余的部分，我们可以继续上面的两步操作，这样，我们就逆序的得到了每一位数，每次求模得到得数就可以组成题目所要求的翻转后的数

示例代码

```
def reverse_int(number):
    reverse_number = 0
    tmp = number
    while tmp > 0:
        end_number = tmp%10   # 得到个位数
        tmp = tmp//10         # 去除个位数后剩余的部分
        reverse_number = reverse_number*10 + end_number  # 组建翻转后的数

    return reverse_number

print(reverse_int(12345))
```

7.12 时间转换

题目要求

请将下列形式的字符串转换成秒数

2小时3分55秒 =》 7435
35分21秒 =》 2121
55秒 =》 55

思路分析

1小时=3600秒
1分钟=60秒
将小时的单位，分钟的单位都转成秒，然后相加，就得到了秒数。可以用"小时"，"分"，"秒"作为分割符分隔字符串，得到时间单位前面的数值。不过这样操作起来，比较麻烦。

我想到一个简单的办法，先将字符串中的"小"字替换掉，然后利用正则表达式模块的split方法分割字符串。

```
import re


string = "1时13分15秒"
arrs = re.split('[时分秒]', string)
print(arrs)
```

程序输出结果为

```
['1', '13', '15', '']
```

1*3600 + 13*60 + 15 = 4395

实例代码

```
import re


def convert_time(time_str):
    seconds = 0
    if not time_str:
        return seconds

    time_str = time_str.replace('小', '')
```

```
    arrs = re.split('[时分秒]', time_str)
    arrs = arrs[:-1]

    base = 1
    for item in reversed(arrs):
        seconds += base*int(item)
        base *= 60
    return seconds


if __name__ == '__main__':
    print(convert_time('2小时3分55秒'))
    print(convert_time('35分21秒'))
    print(convert_time('55秒'))
```

7.13 字典里的value (2)

题目要求

有一个字典，保存的是学生各个编程语言的成绩，内容如下

```
data = {
        'python': {'上学期': '90', '下学期': '95'},
        'c++': ['95', '96', '97'],
        'java': [{'月考':'90', '期中考试': '94', '期末考试': '98'}]
    }
```

各门课程的考试成绩存储方式并不相同，有的用字典，有的用列表，但是分数都是字符串类型，请实现函数transfer_score(score_dict)，将分数修改成int类型

思路分析

不同于上一篇，本次字典的结构变得复杂了，但思路不变，仍然要遍历字典，只是在遍历时，要根据value的类型决定如何进行处理，如果value的类型是字典，那么则仍然按照上一篇的方法进行处理，如果value的类型是列表，则需要对列表里的元素类型进行判断，如果是字符串，则直接转换，如果是字典，则需要按照上一篇的方法进行处理，这次，我采用递归函数进行处理。

示例代码

```python
import pprint


def transfer_score(data):
    # 如果data是字典
    if isinstance(data, dict):
        for key, value in data.items():
            data[key] = transfer_score(value)
        return data

    # 如果data 是列表
    if isinstance(data, list):
        data_lst = []
        for item in data:
            data_lst.append(transfer_score(item))

        return data_lst

    if isinstance(data, str):
        return int(data)


if __name__ == '__main__':
    data = {
        'python': {'上学期': '90', '下学期': '95'},
        'c++': ['95', '96', '97'],
        'java': [{'月考': '90', '期中考试': '94', '期末考试': '98'}]
    }

    data = transfer_score(data)
    pprint.pprint(data)
```

7.14统计代码行数

题目要求

实现函数get_py_program_count，统计指定目录下所有的python文件代码行数

```python
def get_py_program_count(py_path):
    pass
```

思路分析

题目要求统计代码函数，但是一个文件里代码有多少行并没有一个明确的定义，这里就可以有自由发挥的空间，我自己的定义是文件里只要不是空行，只要不是以 # 开头，就算一行有效代码，读取一个文件，逐行进行判断，如果是有效代码，代码行数加1

先实现一个函数，统计一个文件里的代码行数，题目要求的是计算指定目录下所有文件代码行数，那么只需要写一个函数，获取指定目录下所有python脚本的目录，然后用for循环逐个文件计算代码行数并累加即可

示例代码

```
import os


def get_file_lst(py_path, suffix):
    """
    遍历指定文件目录下的所有指定类型文件
    :param py_path:
    :return:
    """
    file_lst = []
    for root, dirs, files in os.walk(py_path):
        for file in files:
            file_name = os.path.join(root, file)
            if file_name.endswith(suffix):
                file_lst.append(file_name)

    return file_lst


def get_program_line_count(file_name):
    """
    返回单个py脚本的代码行数
    :param filename:
    :return:
    """
    # 存储代码行数
    count = 0
    with open(file_name, 'r', encoding='utf-8')as f:
        for line in f:
            line = line.strip()
            if not line:
                continue

            if line.startswith("#"):
                continue
```

```
            count += 1
    return count



def get_py_program_count(py_path):
    # 获取指定目录下所有的python脚本
    file_lst = get_file_lst(py_path, '.py')
    count = 0
    # 遍历列表,获取单个python脚本的代码行数
    for file_name in file_lst:
        count += get_program_line_count(file_name)  # 累加代码函数

    return count   # 返回代码总函数

if __name__ == '__main__':
    py_path = '/Users/kwsy/PycharmProjects/pythonclass'
    print(get_py_program_count(py_path))
```

7.15 提取单词

题目要求

从字符串里提取单词，例如”this is a book“，将单词放到列表里，要求是不能使用split函数

思路分析

字符串相关的算法，往往都需要对字符串进行遍历，设置一个标识位 b_start，初始值设置为False，表示遍历过程中还没有遇到单词

遍历过程中，遇到字母后，b_start修改为True,记录单词开始的位置，遇到空格后，b_start修改为False，记录单词结束的位置，根据开始位置和结束位置进行字符串截取，即可获得单词

示例代码

```
string = " this is a book"
lst = []
# 记录单词开始和结束的位置
b_start = False
start = 0  # 单词开始的位置
end = 0    # 单词结束的位置

for index, item in enumerate(string):
```

```
    if item != " ":
        if b_start:
            continue
        else:
            b_start = True
            start = index
    else:
        if b_start:
            b_start = False
            end = index - 1
            lst.append(string[start:end+1])

if b_start:
    lst.append(string[start:])

print(lst)
```

7.16 学生成绩分析

题目要求

文件score.txt中存储了学生的考试信息，内容如下

```
小明,98
小刚,90
小红,91
小王,98
小刘,80
```

请写代码，读取文件数据，并进行如下分析

1. 最高分和最低分分别是多少?
2. 得最高分的学生有几个？ 得最低分的学生有几个
3. 平均分是多少?

思路分析

读取文件，这没啥可说的，剩下的是简单的统计，不要被文件迷惑，你读取数据以后，转换成列表，不就是你所熟悉的事物了么，如果列表还不熟悉，那你应该好好复习一下列表了

示例代码

```python
def read_file(filename):
    """
    读取文件数据
    :param filename:
    :return:
    """
    f = open(filename, 'r', encoding='utf-8')
    datas = f.readlines()
    datas = [item.strip() for item in datas]
    f.close()

    return datas


def analse_score():
    datas = read_file('score.txt')
    score_lst = []

    for item in datas:
        score = int(item.split(',')[1])
        score_lst.append(score)

    max_score = max(score_lst)
    min_score = min(score_lst)

    max_score_count = score_lst.count(max_score)
    min_score_count = score_lst.count(min_score)

    avg = sum(score_lst)/len(score_lst)

    print(max_score, min_score)
    print(max_score_count, min_score_count)
    print(avg)

if __name__ == '__main__':
    analse_score()
```

7.17 学生成成绩分析

题目要求

文件score.txt中存储了学生的考试信息，内容如下

```
小明,98,96
小刚,90,94
小红,90,94
小王,98,96
小刘,80,90
小赵,90,96
```

第二列是数学成绩，第三列是语文成绩
请写程序分析：

1. 哪些同学语文成绩是相同的?
2. 哪些同学数学成绩是相同的?
3. 哪些同学语文和数学成绩都是相同的?
4. 总分最高的同学是谁，分数是多少?
5. 总分的平均分是多少?
   ### 思路分析
   都是简单的数据统计分析，谈不上思路，跟着代码过一遍，相信你可以理解代码的意图和作用
   ### 示例代码

```
def read_file(filename):
    """
    读取文件数据
    :param filename:
    :return:
    """
    f = open(filename, 'r', encoding='utf-8')
    datas = f.readlines()
    datas = [item.strip() for item in datas]
    f.close()

    return datas


def analse_score():
    datas = read_file('score.txt')
    stu_lst = []

    for data in datas:
        arrs = data.split(',')
        name = arrs[0]
        shuxue = int(arrs[1])
        yuwen = int(arrs[2])
```

```
        stu_lst.append({'name': name, 'shuxue': shuxue, 'yuwen': yuwen})

    yuwen_dict = {}
    shuxue_dict = {}
    ys_dict = {}
    max_score_sum = 0
    stu_score_sum = 0

    for stu in stu_lst:
        name = stu['name']
        shuxue = stu['shuxue']
        yuwen = stu['yuwen']
        score_sum = shuxue + yuwen
        stu_score_sum += score_sum

        if score_sum > max_score_sum:
            max_score_sum = score_sum

        yuwen_dict.setdefault(yuwen, [])
        yuwen_dict[yuwen].append(name)

        shuxue_dict.setdefault(shuxue, [])
        shuxue_dict[shuxue].append(name)

        ys_dict.setdefault((shuxue, yuwen), [])
        ys_dict[(shuxue, yuwen)].append(name)

    for yuwen, name_lst in yuwen_dict.items():
        if len(name_lst) > 1:
            print(yuwen, name_lst)

    print("*"*20)

    for shuxue, name_lst in shuxue_dict.items():
        if len(name_lst) > 1:
            print(shuxue, name_lst)

    print("*"*20)
    for score, name_lst in ys_dict.items():
        sum_score = sum(score)
        if sum_score == max_score_sum:
            print("总分最高的学生", name_lst, score)
        if len(name_lst) > 1:
            print(score, name_lst)

    print(round(stu_score_sum/len(stu_lst), 2))
```

```
if __name__ == '__main__':
    analse_score()
```

7.18 投票选班长

题目要求

文件data里存储了投票选班长的选票情况

```
小红
小刚
小明
小明
小刚
小刘
小红
小张
小王
小明
```

请写代码分析

1. 一共有多少人参加了班长选举?
2. 这些人分别得了多少选票?
   ### 思路分析
   统计有多少人参加了班长选举，将名字存储到集合中，集合的大小就是答案

统计分别得了多少票，先创建一个空字典，以人名做key，0做value,遍历字典，增加value的值，最终就可以得到选举情况

示例代码

```
def read_file(filename):
    """
    读取文件数据
    :param filename:
    :return:
    """
    f = open(filename, 'r', encoding='utf-8')
    datas = f.readlines()
```

```
    datas = [item.strip() for item in datas]
    f.close()

    return datas


def toupiao():
    datas = read_file('data')
    info = {}
    for name in datas:
        if name not in info:
            info[name] = 1
        else:
            info[name] += 1

    print(len(info))
    print(info)


def toupaio2():
    datas = read_file('data')
    info = {}
    for name in datas:
        if name not in info:
            info[name] = 0

        info[name] += 1

    print(len(info))
    print(info)


def toupiao3():
    datas = read_file('data')
    info = {}
    for name in datas:
        info.setdefault(name, 0)
        info[name] += 1

    print(len(info))
    print(info)


if __name__ == '__main__':
    toupiao3()
```

7.19 水果账单

题目要求

文件fruits_saturday.txt中保存了小明周六买水果的消费情况

```
苹果 30.6
香蕉 20.8
葡萄 15.5
草莓 30.2
樱桃 43.9
```

请写程序分析

1. 这些水果一共花费了多少钱?
2. 花费最多的水果是哪个

文件fruits_sunday.txt 中保存了小明周日买水果的消费情况

```
苹果 20.6
香蕉 16.8
樱桃 30.5
大鸭梨 10.5
火龙果 18.8
```

请写程序分析

1. 周六，周日两天，一共买了多少水果
2. 周六，周日两天都买的水果有哪些?
3. 哪些水果是周日买了而周六没有买的?
4. 两天一共花费了多少钱买水果
5. 哪个水果花费的钱最多，花了多少?

思路分析

首先，需要写一个函数，读取账单数据，以字典形式返回消费数据，接下来，则是各种统计操作，需要数量的使用字典，集合这两种数据类型

示例代码

```
def read_fruits(filename):
```

```
    info = {}
    with open(filename, 'r', encoding='utf-8') as f:
        for line in f:
            arrs = line.strip().split()
            name = arrs[0]
            money = float(arrs[1])
            info[name] = money

    return info

def func1():
    info = read_fruits('fruits_saturday.txt')
    # 这些水果一共花费了多少钱?
    values = list(info.values())
    print(sum(values))

    # 花费最多的水果是哪个
    max_money = 0
    name = ''

    for key, value in info.items():
        if value > max_money:
            max_money = value
            name = key

    print(name)


def func2():
    info1 = read_fruits('fruits_saturday.txt')
    info2 = read_fruits('fruits_sunday.txt')

    set1 = set(list(info1.keys()))
    set2 = set(list(info2.keys()))
    # 周六，周日两天，一共买了多少水果
    print(set1.union(set2))

    # 周六，周日两天都买的水果有哪些?
    print(set1.intersection(set2))

    # 哪些水果是周日买了而周六没有买的?
    print(set2.difference(set1))

    # 两天一共花费了多少钱买水果
    print(sum(list(info1.values())) + sum(list(info2.values())))
```

```
    # 哪个水果花费的钱最多，花了多少?
    name = ''
    max_money = 0
    for key, value in info1.items():
        if value > max_money:
            max_money = value
            name = key

    for key, value in info2.items():
        if value > max_money:
            max_money = value
            name = key
    print(name)

if __name__ == '__main__':
    func1()
    func2()
```

7.20 文件读取解析

已知一个文件名为ip.txt的文件，里面存储了大量ip地址

```
192.168.0.1
192.168.0.2
192.168.0.3
192.168.0.1
192.168.0.4
192.168.0.5
192.168.0.5
192.168.0.2
192.168.0.2
192.168.0.5
```

请编写函数，读取文件并分析数据，根据ip出现次数进行排序，程序最终输出ip 和 出现次数，从小到大

思路分析

读取数据，统计每一个ip出现的次数，将统计结果存储到字典中，处理文件中的数据时，时刻要记住，文件中的每一行数据最后都会有 \n 换行符，一定要用strip函数将这个换行符去掉

我们无法对字典进行排序，所以，遍历字典，将字典中的数据存储到列表中，每一项都是一个元组，元组内存储ip 和 出现次数，内容如下

```
[('192.168.0.1', 2), ('192.168.0.2', 3), ('192.168.0.3', 1), ('192.168.0.4', 1), ('
192.168.0.5', 3)]
```

对列表进行排序，可以指定排序的key，这里使用lambda表达式，取列表里每一个元组内的第二个元素做为排序比较时的key

示例代码

```
def analyse_ip(filename):
    ip_info = {}
    with open(filename) as f:
        for line in f:
            ip = line.strip()
            ip_info.setdefault(ip, 0)
            ip_info[ip] += 1

    lst = []
    for key, value in ip_info.items():
        lst.append((key, value))

    lst.sort(key=lambda item:item[1])
    for item in lst:
        print(item)


if __name__ == '__main__':
    analyse_ip('ip.txt')
```

7.21 有效电话号码

题目要求

给定一个包含电话号码列表（一行一个电话号码）的文本文件，假设一个有效的电话号码必须满足以下两种格式：（xxx) xxx-xxxx 或 xxx-xxx-xxxx。（x 表示一个数字）

文本文件里的内容如下：

```
987-123-4567
123 456 7890
(123) 456-7890
(125) 902-2934
```

```
987-123 4997
```

请读取文件并输出符合要求的电话

思路分析

对于电话，email，身份证等信息，有着明确的格式要求，这类信息都可以用正则表达式来判断是否符合格式要求

python里正则表达式要用到re这个模块，先创建一个pattern

```
pattern = re.compile("\d{3}-\d{3}-\d{3}|\(\d{3}\) \d{3}-\d{3}")
```

对于文件里的一行内容，则使用pattern.match方法来判断该行内容是否匹配正则表达式，如果符合则返回一个_sre.SRE_Match 类型的对象，如果不匹配，则返回None

示例代码:

```
# coding=utf-8
import re

# 定义正则表达式
pattern = re.compile("\d{3}-\d{3}-\d{3}|\(\d{3}\) \d{3}-\d{3}")

print type(pattern.match("987-123-4567"))
with open('phones')as fileobj:
    lines = fileobj.readlines()
    for line in lines:
        if pattern.match(line):
            # strip函数去掉line末尾的换行符
            print(line.strip())
```

7.22 旋转数组

题目要求

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数，要求使用空间复杂度为 O(1) 的原地算法

```
输入: [1,2,3,4,5,6,7] 和 k = 3
```

```
输出: [5,6,7,1,2,3,4]
```

思路分析

思路1，多次移动

如果没有空间复杂度为O(1)的要求，这个题目非常简单，简单的做一下切片处理就可以了

```
lst = [1, 2, 3, 4, 5, 6, 7]
lst = lst[len(lst)-3:] + lst[0:len(lst)-3]
print(lst)
```

但切片的过程就是一个复制的过程，我们使用了O(n)的空间，与题目要求不符。

我们只能使用一个O(1)的空间来临时存储数据，简单说，一次只能存储一个数字，但要向右移动K个位置。如果我们能找到一种算法，这个算法执行一次，只将数组向右移动1位，移动k个位置只需要执行这个算法K次就可以了。

考虑移动1位就简单了，用一个临时变量tmp将lst[-1]保存下来，然后从后向前，逐个向后移动，最后，将lst[0]赋值为tmp

```
tmp = lst[-1]
for i in range(len(lst)-1, 0, -1):
    lst[i] = lst[i-1]
lst[0] = tmp
```

经过上面的算法，数组就从[1,2,3,4,5,6,7]变成了[7,1,2,3,4,5,6],如果希望将数组向右移动K个位置，只需要执行K次就可了

```
def move_ele_1(lst, k):
    for j in range(k):
        tmp = lst[-1]
        for i in range(len(lst)-1, 0, -1):
            lst[i] = lst[i-1]
        lst[0] = tmp

if __name__ == '__main__':
    lst = [1, 2, 3, 4, 5, 6, 7]
    move_ele_1(lst, 3)
    print(lst)
```

思路2， 原地翻转

将数组分为两个区域，第一个区域是从0到len(lst)–k–1, 余下的是第二个区域，两个区域分别做一次翻转，然后将整个数组做一次翻转，算法的过程如下

```
[1, 2, 3, 4, 5, 6, 7]   原始数据
[4, 3, 2, 1, 5, 6, 7]   第一个区域翻转
[4, 3, 2, 1, 7, 6, 5]   第二个区域翻转
[5, 6, 7, 1, 2, 3, 4]   整体翻转
```

一旦有了思路，算法就很容易写出来，但要注意处理边界条件

```
def reverse_lst(lst, start, end):
    for i in range(0, (end-start)//2 + 1):
        tmp = lst[start+i]
        lst[start+i] = lst[end-i]
        lst[end-i] = tmp


def move_ele_2(lst, k):
    split_index = len(lst) - k - 1
    reverse_lst(lst, 0, split_index)
    print(lst)
    reverse_lst(lst, split_index+1, len(lst)-1)
    print(lst)
    reverse_lst(lst, 0, len(lst)-1)

if __name__ == '__main__':
    lst = [1, 2, 3, 4, 5, 6, 7]
    move_ele_2(lst, 3)
    print(lst)
```

7.23 合并文件

题目要求

已知一个列表里存储了若干个文件的名字

```
lst = ['1.txt', '2.txt', '3.txt']
```

请写程序，将列表里的文件内容合并到一个文件中，文件命名为'all.txt'

思路分析

写一个for循环，遍历列表lst，逐个打开文件，逐行读取文件里的信息，将读取的每一行数据写入到all.txt文件中，就是这么简单

示例代码

```
lst = ['1.txt', '2.txt', '3.txt']

f = open('all.txt', 'w')

for filename in lst:
    r_file = open(filename)
    for line in r_file:
        f.write(line.strip() + '\n')

    r_file.close()

f.close()
```

7.24 拆分文件

题目要求

已知一个文件名为"all.txt",其内容为

```
2222
111
111
444
3333
6666
8888
```

请写程序拆分这个文件，文件里的内容，以1开头的，就写入到1.txt文件中，以2开头的内容，写入到2.txt文件中，依次类推

思路分析

打开文件，逐行读取文件内容，解析出一行内容的第一个字符，用第一个字符串拼接处要写入的文件名字，然后以a+方式打开文件，将这一行文件写入到文件中

示例代码

```
f = open('all.txt', 'r')

for line in f:
    start = line[0]
    filename = start + ".txt"
    with open(filename, 'a+') as w_file:
        w_file.write(line.strip()+"\n")

f.close()
```

8 地狱难度算法练习题

8.1 删除有序序列中的重复项

题目要求

已知一个有序序列,请原地删除序列中重复出现的元素,返回删除重复元素后的序列长度。

只能使用 O(1) 额外空间来完成这个任务，例如 [0,0,1,1,1,2,2,3,3,4,4,4,5]，最终返回的长度是6, 序列前6个元素是 0 1 2 3 4 5

思路分析

已知条件如下：

- 序列有序
- 只能使用O(1)额外空间，必须原地删除
- 需要返回删除重复元素后的长度，如果长度为N，那么序列前N个元素没有重复元素

设置一个哨兵index，初始值赋为0。用一个for循环从索引1开始遍历lst，i做迭代变量， 如果lst[i]!=lst[index]，就说明lst[i]是一个比lst[index]大的元素，它应该被放置在index+1的位置上

将lst[i] 放置在index+1位置上以后，要向右移动哨兵的位置，执行index += 1，这样，即便lst[i]这个值在后面重复出现了，可此时lst[index]=lst[i]，所以重复出现的值会被忽略掉，直到遇到一个与lst[index]不相等的元素，再次移动哨兵

示例代码

```
def remove_duplicate(lst):
    index = 0
    for i in range(1, len(lst)):
        if lst[index] != lst[i]:
            lst[index + 1] = lst[i]
            index += 1
    return index + 1

if __name__ == '__main__':
    lst = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4, 4, 4, 5]
    index = remove_duplicate(lst)
    print index
    print lst[:index]
```

8.2 组合总数

题目要求

已知一个无重复元素的序列,给定一个目标数,找出序列中所有可以使数字和未目标数的组合。

序列中的元素可以被多次选用,不能出现重复的组合, 序列中的元素和目标数都是正整数。

例如序列 [2, 3, 5]， 目标值为8， 最终的组合有
(2, 3, 3)
(3, 5)
(2, 2, 2, 2)

思路分析

编程不是拿着画笔随心所欲的在画板上涂抹，也不存在固定的方法帮你完成具体的问题，做练习题的目的是通过这些练习题锻炼你的思维，当你建立起编程的思维以后，你也就不在乎什么具体方法和套路，面对具体问题时，你将有能力进行分析。

当问题复杂无从下手时，你可以从问题的边界处入手，题目没有对组合里的元素个数做限制，那么你就先考虑组合里只有一个元素的情况，题目就变成了从序列中找到1个元素，这个元素的值等于目标元素，这样，问题不就变得简单了么

接下来考虑两个元素的情况，从序列中找到两个数，这两个数的和等于目标数target，先随便从序列中选定一个数，假设这个数是i, 那么接下来要做的就是从序列中找到1个元素且这个元素等于target – i

接下来思考三个元素的情况，从序列中找到三个数，这三个数的和等于目标数target，你可以

先随便从序列中选定一个数，假设这个数是i，那么接下来要做的就是从序列中找到两个元素且这两个元素的和等于target – i

新的问题，总是转化为老的问题，而最老的那个问题，从序列中找到一个元素且这个元素的值等于目标值是非常容易解决的。

示例代码

```
# coding=utf-8

def combination_sum(lst, target):
    combination_lst = []
    for item in lst:
        if item == target:
            combination_lst.append([item])
        elif item > target:
            continue   # 单个元素都大于target,这个元素放在哪个组合里,组合之和都必然比ta
rget大
        else:
            other = target - item
            res_lst = combination_sum(lst, other)   # 新问题转化成老问题
            # res_lst 是list, 里面的元素还是list,且和等于other
            for tmp_lst in res_lst:
                tmp_lst.append(item)   # 加上item后,组合之和等于target
            combination_lst.extend(res_lst)   # 所有组合都放在combination_lst

    return combination_lst


def remove_duplicate(lst):
    combination_set = set()
    for item in lst:
        item.sort()  # 先排序
        combination_set.add(tuple(item))   # list 没有hash值,不能放到set中

    return combination_set

if __name__ == '__main__':
    lst = [2, 3, 5]
    res_lst = remove_duplicate(combination_sum(lst, 8))
    for item in res_lst:
        print(item)
```

8.3 不同的子序列(超出时间限制)

题目要求

给定一个字符串S和一个字符串T，计算S的子序列中T出现的次数。
一个串的子串是指该串的一个连续的局部，如果不要求连续,则可称为它的子序列。例如abcde, acd就是abcde的一个子序列
假设 S = "rabbbit", T = "rabbit"，那么有 3 种可以从 S 中得到 "rabbit" 的方案。
(上箭头符号 ^ 表示选取的字母)

rabbbit
^^^^ ^^

rabbbit
^^ ^^^^

rabbbit
^^^ ^^^

思路分析

当分析问题遇到阻碍，我最喜欢从问题的边界处入手，因为边界的地方正是条件达到极值的时候，这时候，很容易就找出破绽。

T的首字母是r， 这个r恰好也是S的首字母，如果让S = "abbbitr" ，让r成为S的末尾字符，问题似乎一下子变得明朗起来，此时，S的任意子序列都不可能是“rabbit“， 因为S中r字符后的后面没有内容了，根本找不到”abbit“，顺着这个思路想下去，想从S的子序列中找到T，就必须满足S中，r的后面可以找到a，a的后面可以找到b，b的后面可以再找到一个b，b的后面可以找打一个i,i的后面可以找到t。

但这样找还是挺麻烦的，为何不把T中每个字符出现在S中的位置记录下来呢？r出现在0的位置上，b出现在2, 3, 4的位置上，用一个字典来保存T中字符在S中出现位置的信息：

```
{
    'a': [
        1
    ],
    'i': [
        5
    ],
    'r': [
        0
    ],
    'b': [
```

```
        2,
        3,
        4
    ],
    't': [
        6
    ]
}
```

T = "rabbit"，上面已经得到每一个字符在S中的位置，i从0到5,根据字符的位置信息挑选T[i]在S中的的合适位置，比如r，它出现的位置信息是[0],只能挑选一个位置，但是b 的位置信息是[2, 3, 4] ,有三个位置可以挑选，需要注意的是，如果T[i]这个字符挑选了位置index，那么T[i+1]这个字符在挑选位置时就不能选小于等于index。假如S=”brabbbit“, 那么在选b的位置的时候，就不能要0这个位置，因为T中a在b的前面，a已经选了2这个位置，b选到0毫无意义，这样不能构成子序列。

分析到这里，就演变成了一个寻找排列组合的问题。

示例代码

```
# coding=utf-8


def num_sequence(source, target):
    # 先得到target的每一个字符在source中出现的位置信息
    index_dict = {}
    for sub_item in target:
        index_dict[sub_item] = []
        for index, item in enumerate(source):
            if sub_item == item:
                index_dict[sub_item].append(index)

    print(index_dict)
    index_seq = get_num_sequence(target, 0, index_dict, -1)
    print(index_seq)

def get_num_sequence(target, t_index, index_dict, s_index):
    """

    :param target: 目标字符串
    :param t_index: 目标字符在target中的位置
    :param index_dict: 目标字符串中每个字符在source中的位置
    :param s_index: 上一个目标字符在S中的位置
    :return:
```

```
    """
    item = target[t_index]
    # 如果目标字符在S中不存在
    if item not in index_dict:
        return []

    # 得到item在S中的位置信息
    index_lst = index_dict[item]
    seq_lst = []
    for index in index_lst:
        # 如果位置比上一个目标字符所选的位置靠前或相等,就不能选
        if index <= s_index:
            continue
        # 如果目标字符已经是最后一个字符了
        if t_index == len(target) - 1:
            seq_lst.append([index])
        else:
            # 递归调用,寻找下一个目标字符可能的位置
            num_seq = get_num_sequence(target, t_index+1, index_dict, index)
            for lst in num_seq:
                lst.insert(0, index)
                seq_lst.append(lst)
    return seq_lst


if __name__ == '__main__':
    source = "aabdbaabeeadcbbdedacbbeecbabebaeeecaeabaedadcbdbcdaabebdadbbaeabdadeaabbabbecebbebcaddaacccebeaeedababedeacdeaaaeeaecbe"
    target = "bddabdcae"
    num_sequence(source, target)
```

8.4逆波兰表达式求值

题目要求

根据逆波兰表示法，求表达式的值。
比如 ["2", "1", "+", "3", "*"]，对应到小学所学的四则运算表达式就是((2 + 1) * 3) 计算结果为9

["4", "13", "5", "/", "+"] 所对应的四则运算表达式是 (4 + (13 / 5)) = 6

现在请计算["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"] 的结果

思路分析

题目所要求计算的逆波兰表达式对应的四则运算表达式是((10 * (6 / ((9 + 3) * -11))) + 17) + 5

逆波兰表达式的计算，其实是栈这种数据结构的经典应用，将逆波兰表达式中的元素逐一放入一个栈中，当遇到运算符时，则从栈顶连续弹出两个元素进行计算，然后把计算结果放入到栈中，最后，栈里只有保存一个元素，这个元素就是整个表达式的计算结果。

因此这道题目，你需要实现一个简单的栈，好在有列表这种神器，栈的实现非常容易。

有一个比较坑的地方是python的除法，python中，采用的是向下取整的办法，比如

- 5/3 = 1.666 向下取整得到1
- 5/-3 = -1.6666 向下取整得到-2

其他语言中，5/-3会向上取整，就会得到-1， 这个是符合我们日常判断的，但python里却偏偏向下取整得到了-2，为了使结果符合人们的常识，需要用到operator 模块的truediv 方法。

示例代码

```
# coding=utf-8
import operator

class Stack(object):
    def __init__(self):
        self.items = []
        self.count = 0

    def push(self, item):
        """
        放入一个新的元素
        :param item:
        :return:
        """
        self.items.append(item)
        self.count += 1

    def top(self):
        # 获得栈顶的元素
        return self.items[self.count-1]

    def size(self):
        # 返回栈的大小
        return self.count
```

```
    def pop(self):
        # 从栈顶移除一个元素
        item = self.top()
        del self.items[self.count-1]
        self.count -= 1
        return item

def cal_exp(expression):
    stack = Stack()
    for item in expression:
        if item in "+-*/":
            # 遇到运算符就从栈里弹出两个元素进行计算
            value_1 = stack.pop()
            value_2 = stack.pop()
            if item == '/':
                res = int(operator.truediv(int(value_2), int(value_1)))
            else:
                res = eval(value_2 + item + value_1)
            # 计算结果最后放回栈,参与下面的计算
            stack.push(str(res))
        else:
            stack.push(item)

    res = stack.pop()
    return res

if __name__ == '__main__':
    print(cal_exp(["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5",
"+"]))
```

8.5 最长公共前缀

题目要求

编写一个函数来查找字符串数组中的最长公共前缀，如果不存在公共前缀，返回空字符串""。
示例1：

```
输入: ["flower","flow","flight"]
输出: "fl"
```

示例2：

```
输入: ["dog","racecar","car"]
输出: ""
```

思路分析

以列表中的第一个单词作为基线，其余的单词逐个字符与第一个单词的字符进行比较，比如以flower做为基线，flower第一个字符是f，那么其余的单词的第一个字符也是f，就将这个f记录下来，逐一比较，直到某个位置上的字符出现不一致的情况。

在比较过程中，要注意单词的长度，如果某个单词已经比较到末尾，那么就可以停止了，因为这个单词的长度就是理论上可能的最大公共前缀

示例代码

```
# coding=utf-8


def get_max_prefix(lst):
    if len(lst) == 0:
        return ""
    base_word = lst[0]
    prefix_lst = []
    for index, item in enumerate(base_word):
        b_common = False     # 标识在index位置上,所有单词的字符是否相同
        for word in lst:
            # index已经超过了单词的长度
            if index >= len(word):
                break

            # 在index这个位置上,字符不相同
            if word[index] != item:
                break
        else:
            # 如果for循环没有被break中断,就会进入到这个语句块
            b_common = True

        if not b_common:
            break
        prefix_lst.append(item)

    return "".join(prefix_lst)
```

```
if __name__ == '__main__':
    lst = ["flower", "flow", "flight"]
    print(get_max_prefix(lst))
```

8.6 最小区间(超出时间限制)

题目要求

已知k 个升序排列的整数数组。找到一个最小区间，使得 k 个列表中的每个列表至少有一个数包含在其中
示例：

```
输入:[[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]
输出: [20,24]
解释:
列表 1：[4, 10, 15, 24, 26]，24 在区间 [20,24] 中。
列表 2：[0, 9, 12, 20]，20 在区间 [20,24] 中。
列表 3：[5, 18, 22, 30]，22 在区间 [20,24] 中。
```

思路分析

列表里有K个元素，这些元素都是列表，下面的分析中，把这K个列表称之为小列表，外层的存储小列表的称之为大列表（big_lst）。题目要求找到一个最小的区间，是的这k个列表中的每个列表至少有一个数包含其中。

我们先随便定义一个区间，比如说[9, 18], 怎么判断这k个列表里每个列表都有元素包含其中呢？这需要遍历每一个列表，列表里的元素只要有一个在9到18之间就可以了。可是这样的遍历好麻烦，因为列表有多个，何不考虑把这k个列表合并成一个呢，这样就只有一个列表存在，寻找区间就不需要对每个列表都检查了。

但是合并区间带来一个新的问题，合并以后，对于一个数，你不知道这个数来自于哪个列表，因此，合并的过程，需要记录每一个数都在哪个列表中出现过，可以使用一个字典（index_dict），用数值做key,value设置为set，存入小列表在大列表中的索引。

经过上面两步分析，可以得到一个合并后的列表，且知道列表中的每个数在哪个小列表中出现过。

接下来就是寻找区间，还是老办法，考虑极值或边界情况，假设合并列表的第一个元素（item）就是最小区间开始的位置，那么只要解决结束位置就好了，要记住，每个元素曾经在哪个小列表中出现过是有记录的，此时可以查看第一个元素在小列表中出现的情况，如果len(index_dict[item]) 和 len(big_lst)相等，就可以证明，item在每个小列表中都出现过。这

样最小区间就是[item, item]。

如果len(index_dict[item]) 和 len(big_lst) 不相等，就看item的下一个元素（next），看它有没有可能是最小区间结束的位置，方法也很简单，把index_dict[item] 和 index_dict[next]做union操作，假设结果为index_set，只要len(index_set) == len(big_lst),那么next就是最小区间结束的位置。如果len(index_set) != len(big_lst)呢，没关系，继续查看下一个，不停的执行index_set = index_dict[item].union(index_dict[next]), 一旦len(index_set) == len(big_lst) 成立，就找到了最小区间的结束位置。

前面的分析，是建立在合并列表的第一个元素是最小区间的起始元素的基础上，依靠这个边界条件，理清的思路，接下来要推广到合并列表的每一个元素，遍历合并列表，并假设当前遍历到的元素就是最小区间的起始元素，然后寻找最小区间结束的元素。

示例代码

```
# coding=utf-8

def get_small_range(lst):
    index_dict = {}     # 记录每一个数在哪个列表中出现过
    merge_lst = []      # 把所有序列合并到一起
    for index, item in enumerate(lst):
        for number in item:
            index_dict.setdefault(number, set([]))
            index_dict[number].add(index)
        merge_lst.extend(item)

    # 合并以后要排序
    merge_lst.sort()
    small_range = None
    small_range_lst = []

    for index, item in enumerate(merge_lst):
        # 获得item 都在哪些列表里出现
        index_set = index_dict[item]
        if len(index_set) == len(lst):
            # 如果item在所有列表里都出现过,直接返回
            return [item, item]
        else:
            # 把item作为区间开始的位置,下面的for循环去寻找区间结束的位置
            for i in range(index+1, len(merge_lst)):
                next = merge_lst[i]
                index_set = index_set.union(index_dict[next])
                # 从item 到 next, 这个范围内每个列表至少有一个元素包含其中
                if len(index_set) == len(lst):
```

```
                    # 记录范围最小的区间
                    if small_range is None or small_range > next - item:
                        small_range = next - item
                        small_range_lst = [item, next]
                    break

    return small_range_lst


if __name__ == '__main__':
    lst = [[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]
    print(get_small_range(lst))
```

8.7森林中的兔子

题目要求

森林中，每个兔子都有颜色。其中一些兔子（可能是全部）告诉你还有多少其他的兔子和自己有相同的颜色。我们将这些回答放在 answers 数组里。

返回森林中兔子的最少数量
示例：

```
示例:
输入: answers = [1, 1, 2]
输出: 5
输入: answers = [10, 10, 10]
输出: 11
```

解题思路

这类编程题，考察的不是编码能力，而是思考能力。首先要明确一点，这些兔子不会撒谎，否则，就没有答案了。

当思考问题时感到无从下手，就考虑边界或极值情况，我们假设森林中所有兔子都是相同的颜色，而且都会告诉你有多少其他的兔子和自己有相同的颜色，那么这会是怎样的局面呢?

假设森林中有3只兔子，那么答案一定是[2, 2, 2]，如果森林中有5只兔子，那么答案一定是[4, 4, 4, 4, 4]。当你限制了那些变化的条件，问题就变得清晰明了了，有N只兔子，就有N个答案，且答案一定是N–1，兔子是不会说谎的。

现在，来分析题目给的例子，[1, 1, 2]，有兔子回答2，就说明有3只兔子是一样的颜色，只有

一只兔子回答了，其他两只没有回答，有两只兔子回答了1，他们两个可能是相同的颜色，也可能是不同的颜色，题目要求算最少的数量，那就认为他们是相同的颜色吧，这样就是5只兔子。

如果答案是 [1, 1, 2, 2, 2, 2]呢，森林中最少有8只兔子，2个回答1的兔子是相同颜色，3个回答2的兔子是相同颜色，还有一只兔子回答2，这就说明还有两个兔子和这只兔子颜色相同，但是他们没有回答。

分析到这里，不难得出一个简单的结论，如果一只兔子回答N，那么森林中必然有N+1只兔子颜色相同，剩余的N只兔子可能来回答了，也可能没有来回答，假设都来回答，那就有N+1个答案为N，可是如果出现了N+2个兔子回答N，就说明又多出来N+1只兔子，多出来的这组兔子只有一只来回答了。

你需要做的是统计每个答案的个数，比如 [1, 1, 2, 2, 2, 2]，回答情况如下

```
{
    1: 2,   # 回答1 的有2只兔子
    2: 4    # 回答2 的有4只兔子
}
```

回答2，表明有3只兔子，但却有4只兔子回答2，用4除以3得1.33，这说明存在两组兔子，每组有3只颜色一样的兔子。

示例代码

```
# coding=utf-8
import math


def get_rabbit_count(answers):
    answer_dict = {}
    for item in answers:
        answer_dict.setdefault(item, 0)
        answer_dict[item] += 1

    count = 0
    for key, value in answer_dict.items():
        # math.ceil向上取整, math.ceil(float(value)/(key+1)) 得到的是至少有几组兔子
        # 有兔子回答key,那么就至少有key+1个兔子
        count += math.ceil(float(value)/(key+1))*(key+1)

    return int(count)
```

```
if __name__ == '__main__':
    print(get_rabbit_count([1, 1, 2]))
    print(get_rabbit_count([10, 10, 10]))
    print(get_rabbit_count([1, 1, 2, 2, 2, 2]))
```

8.8最大子序和

一个列表里，都是整数，请到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

例如列表 [−2,1,−3,4,−1,2,1,−5,4]， 其连续子数组[4,−1,2,1]的和最大为6.

思路分析

动态规划的算法，难与易只在一念之间，掌握了动态规划的精髓，代码轻易的就能写出，反之，则陷入到茫然之中。

动态规划的题目，其实，都可以用暴力破解，但真正的解法都非常的巧妙，说是巧妙，并不是投机取巧，而是掌握了动态规划的实质。

既然是动态，那么到底是什么在动呢？假设数组为k，令max_sum表示最大子数组之和，已知从k[i]到[j]的的和为pre_sum，暂且让max_sum = pre_sum，那么现在，我们要考虑是否应该让k[j+1]也加到这个子数组之中，多一个数，可能让max_sum更大哦。

如果pre_sum >=0 ,那么就让pre_sum += k[j+1]，你一定会提出质疑，如果k[j+1]是负数，那么子数组之和不是更小了么？这里你忽略了，我已经让max_sum=pre_sum，所以我已经记录了最大值，pre_sum+= k[j+1]之后，的确变小了，但是max_sum还是之前的那个最大值，之所以要加k[j+1]，因为k[j+2]有可能是一个很大的正数啊，所以要继续探索可能的最大子数组之和。那么假如k[j+2]也是负数呢，而且是很大的负数，以至于加上k[j+2]之后，pre_sum<0，如果这样的事情发生，就按下面的方法操作

如果pre_sum < 0,不论k[j+1]是正还是负，k[j+1]加上pre_sum以后所得到的和都一定比k[j+1]更小，而我们要的是最大子序之和，所以这时，应该让max_sum等于pre_sum和k[j+1]中最大的那个，同时，让pre_sum = k[j+1]，表示重新开始寻找和最大的子数组，如果不重新开始，带着pre_sum这个负数，难道对求和不是一个负面效果么。

谁在动呢，是pre_sum一直在动，我们规划的是一个子数组，期初，这个子数组里只有一个元素，就是数组的第一个元素k[0],随后就是考虑要不要把k[1]加进来，是否加进来，就按照上面的说的方法来进行。

示例代码

```python
# coding=utf-8

def max_sub_sum(lst):
    max_sum = lst[0]
    pre_sum = lst[0]   # pre_sum是动态的,最初等于列表的第一个元素
    for i in range(1, len(lst)):
        # 前面的累积和如果小于0,当前值item加上一个负数只会比item更小
        # 因此将item赋值给pre_sum
        if pre_sum < 0:
            pre_sum = lst[i]
        else:
            # 前面的累积和是整数或者0,继续累加
            pre_sum += lst[i]

        if pre_sum > max_sum:
            max_sum = pre_sum

    return max_sum


if __name__ == "__main__":
    print max_sub_sum([-2, 1, -3, 4, -1, 2, 1, -5, 4])
```

小结

既然是动态规划，就要找到那个变化的点，就这道题目而言，变化的点就是连续子数组之和小于0了，这个时候就必须从新开始寻找了，因为这个和已经是负数了，就如同一个包袱，就算后面的数都是正数，加上一个负数也终究比都是整数要小。

8.8 简单的计算器

题目要求

实现一个简单的计算器，能够计算简单字符串表达式，表达式可以包括小括号，运算符包含+ – * /, 表达式允许有空格,参与运算的数值都是正整数。
示例1

```
输入: "1 + 2"
输出: 3
```

示例2

```
输入:" 2 - 3 + 2 "
输出: 1
```

示例3

```
输入:"(1+(4+5+3)-3)+(9+8)"
输出: 27
```

示例4

```
输入:"(1+(4+5+3)/4-3)+(6+8)*3"
输出: 43
```

思路分析

1 表达式预处理

万事开头难，先不考虑如何计算，我们应该先对表达式进行处理，处理以后，只有数值和运算符，这样才能对他们进行具体的操作，比如"1 + 2" 处理后得到['1', '+', '2']， 运算符和数值都分离开了。

这样的解析并不复杂，只需要遍历字符串，解析出数值部分放入到列表中，遇到小括号或者运算符则直接放入列表中，代码如下：

```python
def exp_to_lst(exp):
    lst = []
    start_index = 0     # 数值部分开始位置
    end_index = 0       # 数值部分结束位置
    b_start = False
    for index, item in enumerate(exp):
        # 是数字
        if item.isdigit():
            if not b_start:  # 如果数值部分还没有开始
                start_index = index     # 记录数值部分开始位置
                b_start = True          # 标识数值部分已经开始
        else:
            if b_start:  # 如果数值部分已经开始
                end_index = index      # 标识数值部分结束位置
                b_start = False        # 标识数值部分已经结束
                lst.append(exp[start_index:end_index])    # 提取数值放入列表
```

```
            if item in ('+', '-', '*', '/', '(', ')'):    # 运算符直接放入列表
                lst.append(item)

    if b_start:    # 数值部分开始了,但是没有结束,说明字符串最后一位是数字,
        lst.append(exp[start_index:])
    return lst

def test_exp_to_lst():
    print exp_to_lst("1 + 2")
    print exp_to_lst(" 2 - 3 + 2 ")
    print exp_to_lst("(1+(4+5+3)-3)+(9+8)")
    print exp_to_lst("(1+(4+5+3)/4-3)+(6+8)*3")

test_exp_to_lst()
```

程序输出结果为

```
['1', '+', '2']
['2', '-', '3', '+', '2']
['(', '1', '+', '(', '4', '+', '5', '+', '3', ')', '-', '3', ')', '+', '(', '9', '+
', '8', ')']
['(', '1', '+', '(', '4', '+', '5', '+', '3', ')', '/', '4', '-', '3', ')', '+', '(
', '6', '+', '8', ')', '*', '3']
```

2 中序转后序

1+2 这种表达式是中序表达式，运算符在运算对象的中间，还有一种表达式，运算符在运算对象的后面，称之为后序表达式，也叫逆波兰表达式。1+2 转成后序表达式后是 1 2 +， +号作用于它前面的两个运算对象。

后序表达式相比于中序表达式更容易计算，因此，这一小节，我要把中序表达式转换成后序表达式。

转换时要借助栈这个数据结构，变量postfix_lst 表示后序表达式，遍历中序表达式，根据当前值进行处理：

- 如果遇到数值，则放入到postfix_lst中
- 遇到（ 压如栈中
- 遇到 ），将栈顶元素弹出并放入到postfix_lst中，直到遇到（，最后弹出（
- 遇到运算符，把栈顶元素弹出并放入到postfix_lst中，直到栈顶操作符的运算优先级小于当前运算符，最后将当前运算符压入栈

代码如下：

```
# 运算优先级
priority_map = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2
}

class Stack(object):
    def __init__(self):
        self.items = []
        self.count = 0

    def push(self, item):
        """
        放入一个新的元素
        :param item:
        :return:
        """
        self.items.append(item)
        self.count += 1

    def top(self):
        # 获得栈顶的元素
        return self.items[self.count-1]

    def size(self):
        # 返回栈的大小
        return self.count

    def pop(self):
        # 从栈顶移除一个元素
        item = self.top()
        del self.items[self.count-1]
        self.count -= 1
        return item

def infix_exp_2_postfix_exp(exp):
    """
    中序表达式转后序表达式
    """
    stack = Stack()
    exp_lst = exp_to_lst(exp)
    postfix_lst = []
    for item in exp_lst:
```

```
        # 如果是数值,直接放入到postfix_lst 中
        if item.isdigit():
            postfix_lst.append(item)
        elif item == '(':
            # 左括号要压栈
            stack.push(item)
        elif item == ')':
            # 遇到右括号了,整个括号里的运算符都输出到postfix_lst
            while stack.top() != '(':
                postfix_lst.append(stack.pop())
            # 弹出左括号
            stack.pop()
        else:
            # 遇到运算符,把栈顶输出,直到栈顶的运算符优先级小于当前运算符
            while stack.size() != 0 and stack.top() in ("+-*/")\
                    and priority_map[stack.top()] >= priority_map[item]:
                postfix_lst.append(stack.pop())
            # 当前运算符优先级更高,压栈
            stack.push(item)

    # 最后栈里可能还会有运算符
    while stack.size() != 0:
        postfix_lst.append(stack.pop())

    return postfix_lst

print infix_exp_2_postfix_exp("1 + 2")
print infix_exp_2_postfix_exp(" 2 - 3 + 2 ")
print infix_exp_2_postfix_exp("(1+(4+5+3)-3)+(9+8)")
print infix_exp_2_postfix_exp("(1+(4+5+3)/4-3)+(6+8)*3")
```

程序输出结果为

```
['1', '2', '+']
['2', '3', '-', '2', '+']
['1', '4', '5', '+', '3', '+', '+', '3', '-', '9', '8', '+', '+']
['1', '4', '5', '+', '3', '+', '4', '/', '+', '3', '-', '6', '8', '+', '3', '*', '+
']
```

3 后序表达式计算

后续表达式计算同样需要用到栈，这个算法在逆波兰表达式计算的练习题中已经有讲解，直接复用代码

```
def cal_exp(expression):
    stack = Stack()
    for item in expression:
        if item in "+-*/":
            # 遇到运算符就从栈里弹出两个元素进行计算
            value_1 = stack.pop()
            value_2 = stack.pop()
            if item == '/':
                res = int(operator.truediv(int(value_2), int(value_1)))
            else:
                res = eval(value_2 + item + value_1)
            # 计算结果最后放回栈,参与下面的计算
            stack.push(str(res))
        else:
            stack.push(item)

    res = stack.pop()
    return res
```

全部代码

```
# coding=utf-8
import operator

# 运算优先级
priority_map = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2
}

class Stack(object):
    def __init__(self):
        self.items = []
        self.count = 0

    def push(self, item):
        """
        放入一个新的元素
        :param item:
        :return:
        """
        self.items.append(item)
```

```
        self.count += 1

    def top(self):
        # 获得栈顶的元素
        return self.items[self.count-1]

    def size(self):
        # 返回栈的大小
        return self.count

    def pop(self):
        # 从栈顶移除一个元素
        item = self.top()
        del self.items[self.count-1]
        self.count -= 1
        return item


# 表达式中序转后序
def infix_exp_2_postfix_exp(exp):
    """
    中序表达式转后序表达式
    """
    stack = Stack()
    exp_lst = exp_to_lst(exp)
    postfix_lst = []
    for item in exp_lst:
        # 如果是数值,直接放入到postfix_lst 中
        if item.isdigit():
            postfix_lst.append(item)
        elif item == '(':
            # 左括号要压栈
            stack.push(item)
        elif item == ')':
            # 遇到右括号了,整个括号里的运算符都输出到postfix_lst
            while stack.top() != '(':
                postfix_lst.append(stack.pop())
            # 弹出左括号
            stack.pop()
        else:
            # 遇到运算符,把栈顶输出,直到栈顶的运算符优先级小于当前运算符
            while stack.size() != 0 and stack.top() in ("+-*/")\
                    and priority_map[stack.top()] >= priority_map[item]:
                postfix_lst.append(stack.pop())
            # 当前运算符优先级更高,压栈
            stack.push(item)
```

```
    # 最后栈里可能还会有运算符
    while stack.size() != 0:
        postfix_lst.append(stack.pop())

    return postfix_lst




def exp_to_lst(exp):
    """
    表达式预处理，转成列表形式
    """
    lst = []
    start_index = 0    # 数值部分开始位置
    end_index = 0      # 数值部分结束位置
    b_start = False
    for index, item in enumerate(exp):
        # 是数字
        if item.isdigit():
            if not b_start:  # 如果数值部分还没有开始
                start_index = index    # 记录数值部分开始位置
                b_start = True         # 标识数值部分已经开始
        else:
            if b_start:  # 如果数值部分已经开始
                end_index = index     # 标识数值部分结束位置
                b_start = False       # 标识数值部分已经结束
                lst.append(exp[start_index:end_index])   # 提取数值放入列表

            if item in ('+', '-', '*', '/', '(', ')'):    # 运算符直接放入列表
                lst.append(item)

    if b_start:    # 数值部分开始了,但是没有结束,说明字符串最后一位是数字,
        lst.append(exp[start_index:])
    return lst


def cal_exp(expression):
    """
    计算后续表达式
    """
    stack = Stack()
    for item in expression:
        if item in "+-*/":
            # 遇到运算符就从栈里弹出两个元素进行计算
```

```
            value_1 = stack.pop()
            value_2 = stack.pop()
            if item == '/':
                res = int(operator.truediv(int(value_2), int(value_1)))
            else:
                res = eval(value_2 + item + value_1)
            # 计算结果最后放回栈,参与下面的计算
            stack.push(str(res))
        else:
            stack.push(item)

    res = stack.pop()
    return res


def test_exp_to_lst():
    print exp_to_lst("1 + 2")
    print exp_to_lst(" 2 - 3 + 2 ")
    print exp_to_lst("(1+(4+5+3)-3)+(9+8)")
    print exp_to_lst("(1+(4+5+3)/4-3)+(6+8)*3")


def test_infix_exp_2_postfix_exp():
    print cal_exp(infix_exp_2_postfix_exp("1 + 2"))
    print cal_exp(infix_exp_2_postfix_exp(" 2 - 3 + 2 "))
    print cal_exp(infix_exp_2_postfix_exp("(1+(4+5+3)-3)+(9+8)"))
    print cal_exp(infix_exp_2_postfix_exp("(1+(4+5+3)/4-3)+(6+8)*3"))


if __name__ == '__main__':
    test_infix_exp_2_postfix_exp()
```

8.9最长上升子序列

题目要求

一个元素为数值的列表，找到其最长上升子序列的长度。

比如 [5, 4, 1, 2, 5, 3]， 最长上升子序列是 [1, 2, 3], 长度为3。

思路分析

假设列表lst长度为k，创建一个长度同样为k的列表dp,dp内所有元素初始化为1，dp[i]代表以

lst[i]结尾的最长上升子序列长度，这其实是一个假设，对于单个元素来说，也是列表的子序列，而且是上升的。

对于lst[0]来说，它自己就是一个子序列，单个元素，也可以视为上升的，因此，dp[0] 等于1，真实的表示了以lst[0]结尾的最长上升子序列的长度，但对于lst[j], j >0 来说，dp[j]就目前而言，还不能准确表示以lst[j]结尾的最长上升子序列的长度，不过没关系，计算dp[1]时，我们可以借助dp[0]，计算dp[2]时，可以借助dp[0]，dp[1]，计算dp[j]时，可以借助dp[0]到dp[j−1]。

计算dp[j]时，如何借助之前已经计算出来的dp[0]到dp[j−1]呢，太简单了，假设 0<=i<j, 如果lst[i] < lst[j], 那么dp[j] = dp[i] + 1，dp[i]是已知的，lst[i] 小于lst[j]， 这个上升子序列加入了lst[j],自然长度加1。

示例代码

```
# coding=utf-8


def get_long_incr_lst(lst):
    if len(lst) == 0:
        return 0
    dp = [1 for item in lst]
    max_incr = 1
    # 为什么要从1 开始呢,因为dp[0]是固定的,一定是1
    for j in range(1, len(lst)):
        for i in range(j):
            if lst[i] < lst[j]:
                # 以lst[i]结尾的上升子序列里可以加入lst[j]
                dp[j] = max(dp[j], dp[i] + 1)
        max_incr = max(dp[j], max_incr)

    return max_incr


print get_long_incr_lst([5, 4, 1, 2, 5, 3])
```

8.10 山脉数组的顶峰索引

题目要求

如果一个数组k符合下面两个属性，则称之为山脉数组

- 数组的长度大于等于3
- 存在i，i >0 且 i < len(k)−1， 使得k[0] < k[1]< ... < k[i−1] < k[i] > k[i+1] ... >k[len(k)−1]， 这个i就是顶峰索引。

现在，给定一个山脉数组，求顶峰索引。

```
输入: [0, 1, 3, 5, 4, 3, 2]
输出: 3
```

思路分析

遍历数组，每个当前元素都和自己右侧的元素比较，如果当前元素大于右侧的元素，则当前这个元素的索引就是山脉数组的顶峰索引

示例代码

```
# coding=utf-8

def get_top_index(lst):
    # 注意起始位置和结束位置
    for i in range(1, len(lst)-1):
        if lst[i] > lst[i+1]:
            return i


if __name__ == '__main__':
    lst = [0, 1, 3, 5, 4, 3, 2]
    print get_top_index(lst)
```

使用二分查找法

上面的算法虽然很简单，但效率并不高，可以使用二分查找法获得更快的速度。二分查找法利用了数组的有序性，山脉数组虽然不是有序的，但是峰顶前和峰顶后却各自有序，前者是升序，后者是降序，那么同样可以利用二分查找法进行查找

代码如下:

```
# coding=utf-8


def get_top_index(lst, start, end):
```

```
    # 中间位置
    middle = (start + end)/2
    # 恰好是峰顶
    if lst[middle] > lst[middle-1] and lst[middle] > lst[middle+1]:
        return middle
    # 还没有到达峰顶
    if lst[middle] < lst[middle+1]:
        return get_top_index(lst, middle+1, end)
    elif lst[middle] > lst[middle+1]:   # 已经过了峰顶
        return get_top_index(lst, start, middle-1)


if __name__ == '__main__':
    lst = [0, 1, 3, 5, 4, 3, 2, 1]
    print get_top_index(lst, 0, len(lst)-1)
```

8.11 两个有序数组中找第k大的数

题目要求

已知有两个从小到大的有序数组，求两个数组的第k大的数。

```
[1, 4, 6, 8, 12, 15, 18, 20, 28, 29]
[2, 5, 7, 10]
第8大的数是10
```

思路分析

两个数组都有序，那么就利用这个有序的特点来解决这个问题。假设数组分别是a b,令middle = k/2， middle_ex = k – middle。比较a[middle]和b[middle]的值。

- 如果a[middle – 1] == b[middle_ex – 1]，那么a[middle–1]不正好是第k大的数么，因为 k= middle_ex + middle,且两个数组都有序。
- 如果a[middle – 1] < b[middle_ex – 1]，让a = a[middle:],前面的那些元素都可以舍弃了， 问题转变成从a 和 b这两个数组里找到第 k – middle 大的值
- 如果a[middle – 1] > b[middle_ex – 1]，让b = b[middle_ex:]，前面那些元素都可以舍弃了，问题转变成从a 和 b这两个数组里找到第 k – middle_ex大的值

为什么在取值时用的索引是middle – 1呢，其实原因很简单，我们要找第k大的数，k最小是1，你不能说取第0大的数，我们日常是从1开始计数的，而计算机是从0开始计数的，middle

= k/2， 从k计算而来，因此在使用索引时要减 1 。

有几个需要关注的地方

1. 要让数组长度更小的为a
2. 计算middle时，其实要考虑middle是否比a的长度小，不然取a[middle-1]时就出错了，计算middle和 middle_ex为的是从a， b 两个数组里各自找到第middle大和第middle_ex大的两个数，通过比较他们的大小，决定舍弃哪一部分
3. 不断的舍弃，不断的修改k的值，最后，一定会出现k==1的情况，这时，返回min(a[0], b[0])

示例代码

```
# coding=utf-8


def find_kth(left_lst, left_len, right_lst, right_len, k):
    """
    从left_lst 和 right_lst中寻找第k大的数
    :param left_lst: 长度小的那个数组
    :param left_len:
    :param right_lst: 长度达的那个数组
    :param right_len:
    :param k:
    :return:
    """
    # 确保left_lst长度小于 right_lst 长度
    if left_len > right_len:
        return find_kth(right_lst, right_len, left_lst, left_len, k)

    # 长度小的数组已经没有值了,从right_lst找到第k大的数
    if left_len == 0:
        return right_lst[k-1]

    # 找到第1 大的数,比较两个列表的第一个元素,返回最小的那个
    if k == 1:
        return min(left_lst[0], right_lst[0])

    # k >> 1 ,其实就是k/2
    middle = min(k >> 1, left_len)
    middle_ex = k - middle
    # 舍弃left_lst的一部分
    if left_lst[middle-1] < right_lst[middle_ex-1]:
```

```
        return find_kth(left_lst[middle:], left_len-middle, right_lst, right_len, k
-middle)
    # 舍弃right_lst 的一部分
    elif left_lst[middle-1] > right_lst[middle_ex-1]:
        return find_kth(left_lst, left_len, right_lst[middle_ex:], right_len-middle
_ex, k-middle_ex)
    else:
        return left_lst[middle-1]


if __name__ == '__main__':
    left_lst = [1, 4, 6, 8, 12, 15, 18, 20, 28, 29]
    right_lst = [2, 5, 7, 10]
    k = 8
    print find_kth(left_lst, len(left_lst), right_lst, len(right_lst), k)

    # 合并后排序,找第k大的数
    lst = left_lst + right_lst
    lst.sort()
    print lst[k-1]
```

8.12 搜索二维矩阵

题目要求

已知一个矩阵，有以下特点：

- 每行的元素从左到右是升序的
- 每列的元素从上到下是升序的

示例如下

```
matrix = [
              [1, 4, 7, 11, 15],
              [2, 5, 8, 12, 19],
              [3, 6, 9, 16, 22],
              [10, 13, 14, 17, 24],
              [18, 21, 23, 26, 30]
            ]
```

请编写一个高效的算法搜索目标值target

思路分析

每行数据和每列数据都是升序的，但行与行之间并没有升序关系，第二行的数据并不是都比第一行大。因此，搜索数据时，必须从第一行开始进行搜索。

搜索某一行时，由于数据有序，因此可以轻松的找到最小值和最大值

- 如果最小值比target大，可以确定整个矩阵里都没有目标值
- 如果最大值比target小，说明这一行里肯定不存在目标值。
- 如果target>= min and target <= max，就在这一行里进行查找，关键之处在于，如果这一行里没有找到目标值，还要继续去下一行里查找么？答案是肯定的，比如你想找5，第一行里没有找到，但是第二行里却可以找得到，而且，你无法跳过第一行直接查找第二行，因为第一行里可能有5

前面的三点分析，已经让搜索算法非常高效了，接下来要考虑如何高效的在一行数据里查找目标值，由于每一行数据是有序的，因此可以使用二分查找法，二分查找法已经非常高效了，但是由于每一列从上到下也是升序的，因此，可以对二分查找法稍稍做一点改变，就可以让整个的查找速度更快。

如果二分查找法在找不到目标值时，返回第一个比目标值大的元素的索引，那么，这个索引就可以作为下一次查找时结束位置的边界，比如搜索9，对第一行使用二分查找法搜索时没有找到目标值，但是找到了第一个比9大的元素11，索引为3，由于每一列都是升序的，所以搜索第二行时，不需要搜索整行，只需要搜索0到2这个索引范围就可以了。通过对二分查找法小做修改，就使得之后的每一行搜索范围得以缩小，效率更高。

示例代码

```
def binary_search(lst, start, end, target):
    """
    如果目标值存在,则返回目标值索引和下一个位置
    如果目标值不存在,则返回第一个比目标值大的元素的索引
    如果最大值比目标值还小,则返回整行的长度
    :param lst:
    :param start:
    :param end:
    :param target:
    :return:
    """
    if start > end:
         return -1, start + 1

    middle = (start + end)//2
    if lst[middle] == target:
        return middle, middle + 1
    elif lst[middle] > target:
        return binary_search(lst, start, middle-1, target)
```

```
    else:
        return binary_search(lst, middle+1, end, target)


def search_matrix(matrix, target):
    if len(matrix) == 0:
        return 0
    if len(matrix[0]) == 0:
        return False

    end_index = len(matrix[0]) -1
    for lst in matrix:
        min_value = lst[0]
        max_value = lst[-1]
        if target < min_value:
            return False  # 此处可以直接返回False
        if target > max_value:
            continue  # 目标值比这一行最大值还大,去下一行查找
        target_index, end_index = binary_search(lst, 0, end_index, target)
        end_index -= 1  # 下一次搜索时结束的位置
        if target_index >= 0:
            return True

    return False


if __name__ == '__main__':
    matrix = [
              [1, 4, 7, 11, 15],
              [2, 5, 8, 12, 19],
              [3, 6, 9, 16, 22],
              [10, 13, 14, 17, 24],
              [18, 21, 23, 26, 30]
             ]
    print(search_matrix(matrix, 6))
    print(search_matrix(matrix, 16))
    print(search_matrix(matrix, 20))
    print(search_matrix([[1]], 1))
```

8.13 判断子序列

题目要求

给定字符串 s 和 t ， 判断 s 是否为 t 的子序列

你可以认为 s 和 t 中仅包含英文小写字母。字符串 t 可能会很长（长度 ~= 500,000），而 s 是个短字符串（长度 <=100）

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）

示例 1:
s = "abc", t = "ahbgdc"
返回 true.

示例 2:
s = "axc", t = "ahbgdc"
返回 false

思路分析

一种思路是从t中寻找长度和s相同的子序列，然后判断这些子序列中有没有和s相等的子序列。这虽然是一个可行的方案，但显然效率很差。

另一个思路是先对t进行分析，遍历t获得每一个字符的出现位置，最终将得到一个字符分布的信息，例如“ahbgdc”，它的字符分布情况是

```
{
'a': [0],
'b': [2],
'c': [5],
'd': [4],
'g': [3],
'h': [1]
}
```

得到这些字符分布情况后，就可以根据每个字符的所在位置来判断字符串s是否是t的子序列，假设a出现的位置都大于b出现的位置，那么字符串s一定不是字符串t的子序列，每个字符可以出现在很多个位置上，我们只要找到一个位置满足s是t的子序列就可以了。

以"abc" 为例，先找到a出现的位置，不论有多少个，只取第一个，记为pre_index，因为a的位置越靠前，留给bc的空间就越大，那么接下来从出现的位置中寻找第一个比pre_index大的位置并赋值给pre_index，为什么要找第一个，因为b的位置越靠前，留给c的空间就越大，最后从c的位置找到第一个比pre_index大的位置

示例代码

```python
def is_sub_seq(s, t):
    char_info = get_char_info(t)
    pre_index = -1
    for item in s:
        # 如果t中不包含item这个字符
        index_lst = char_info.get(item)
        if index_lst is None:
            return False
        for index in index_lst:
            # pre_index是上一个字符可以出现的位置
            # 当前字符出现的位置中,找第一个比pre_index大的位置
            if index > pre_index:
                pre_index = index
                break
        else:
            # 如果循环正常退出
            return False

    return True


def get_char_info(t):
    info = {}
    for index, item in enumerate(t):
        info.setdefault(item, [])
        info[item].append(index)
    return info


if __name__ == '__main__':
    print(is_sub_seq("abc", "ahbgdc"))
```

8.14 最长重复子数组

题目要求

给两个整数数组 A 和 B ， 返回两个数组中公共的、长度最长的子数组的长度。

```
输入:
A: [1,2,3,2,1]
B: [3,2,1,4,7]
输出: 3
```

题目分析

这个题目本质上就是最长公共子串算法，使用动态规划算法就可以轻松解决。

假设A[i] == B[j], 这时，我们不去考虑A[i+1]好B[j+1]的关系，而是考虑A[i−1]和B[j−1]的关系。动态规划，总是依据前面的结果来计算当前的结果。

那么要依据前面的什么结果呢？假设A[i]和B[j]是最长公共子串的结尾处，那么我们用DP[i+1][j+1]记录这个子串的长度。 现在，请考虑DP[i][j]表达的含义是什么呢？ 它表示以A[i−1]和B[j−1]结尾的公共子串的长度，且DP[i][j] + 1 = DP[i+1][j+1]。

为什么要用DP[i+1][j+1]来记录以A[i]和B[j]结尾的公共子串的长度呢？看起来，下角标不一致，感觉怪怪的。

之所以这样做，是因为动态规划总是用前面的结果来计算当前的结果。当我们计算DP[i+1][j+1]时，需要用到DP[i][j]，i和j都是从0开始的。当我们考虑A[0]和B[0]的关系时，我们想要计算的是DP[1][1]，这时，我们就可以利用DP[0][0],如果用DP[0][0]表示以A[0],B[0]结尾的公共子串的长度，那么我们就不得不利用DP[−1][−1]+1来计算DP[0][0]，但是DP[−1][−1]并不存在。

因此，DP这个二维矩阵的第一行和第一列都设置为0，如此，就可以完全使用DP[i][j] + 1 = DP[i+1][j+1] 这个公式来进行计算，如下图所示

		a	b	c	d	e	f	g
	0	0	0	0	0	0	0	0
x	0	0	0	0	0	0	0	0
y	0	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0	0
a	0	1	0	0	0	0	0	0
b	0	0	2	0	0	0	0	0
c	0	0	0	3	0	0	0	0
d	0	0	0	0	4	0	0	0

示例代码

```
def find_max_lengh(lst_1, lst_2):
    max_len = 0
    dp = [[0 for i in range(len(lst_1)+1)] for j in range(len(lst_2)+1)]
    for i in range(len(lst_1)):
        for j in range(len(lst_2)):
            if lst_1[i] == lst_2[j]:
                dp[i+1][j+1] = dp[i][j] + 1
                if dp[i+1][j+1] > max_len:
                    max_len = dp[i+1][j+1]

    return max_len
```

```
if __name__ == '__main__':
    lst_1 = [1, 2, 3, 2, 1]
    lst_2 = [3, 2, 1, 4, 7]
    max_len = find_max_lengh(lst_1, lst_2)
    print(max_len)
```

8.15 找到 K 个最接近的元素

题目要求

给定一个排序好的数组，两个整数 k 和 x，从数组中找到最靠近 x（两数之差最小）的 k 个数。返回的结果必须要是按升序排好的。如果有两个数与 x 的差值一样，优先选择数值较小的那个数

```
输入: [1, 5, 8, 9, 17, 20], k=4, x=10
输出: [5, 8, 9, 17]
```

思路分析

第一个大于等于目标值的元素位置

题目要求找到K个最接近目标值的元素，我们先不考虑K个，而是考虑1个的情况，也就是最接近目标值的那个元素。

如果数组里有某个元素恰好和目标值相等，那么问题就转变成了二分查找法，在数组中查找目标值。如果数组里没有目标值呢，如何找到距离目标值最近的元素呢?

不妨先找到第一个大于等于目标值的元素，找到了这个元素，也就容易找到距离目标值最近的那个元素了，分析到这里，我们先要实现一个函数，该函数可以从数组中找到第一个大于等于目标值的元素位置，下面是这个函数的示例代码

```
def find_eg_index(lst, start, end, target):
    """
    找到第一个大于等于target的元素的位置，如果lst中最大的元素小于target
    则返回len(lst)
    :param lst:
    :param start:
    :param end:
```

```
    :param target:
    :return:
    """
    if start > end:
        return end + 1
    middle = (start + end)//2
    middle_value = lst[middle]
    if middle_value == target:
        return middle
    elif middle_value < target:
        return find_eg_index(lst, middle+1, end, target)
    else:
        return find_eg_index(lst, start, middle-1, target)
```

一定要记得考虑边界条件，如果目标值比数组最小值还小，那么函数返回0是应当的，但如果目标值比数组中最大值还大，该返回什么呢?为了概念的完整性，我这里返回len(lst), 这个索引是超出数组范围的，这表示无穷大，无穷大当然大于等于target

距离目标值最近的元素位置

前面的find_eg_index函数返回了第一个大于等于target的元素的位置eg_index，在此基础上，很容易就算出来距离target最近的元素的位置，所要考虑的逻辑分支如下

- eg_index == len(lst), target > max(lst), 数组最后一个元素最接近target
- eg_index == 0, 数组第一个元素最接近target
- target – lst[eg_index–1] <= lst[eg_index] – target, 说明lst[eg_index–1]更接近target, 反之lst[eg_index]更接近target

实现函数find_closest_index(lst, target) ，返回数组中距离target最近的元素的位置

```
def find_closest_index(lst, target):
    """
    寻找距离target最近的位置
    :param lst:
    :param target:
    :return:
    """
    eg_index = find_eg_index(lst, 0, len(lst)-1, target)
    if eg_index == len(lst):
        return eg_index - 1
    elif eg_index == 0:
        return 0
    else:
        if target - lst[eg_index-1] <= lst[eg_index] - target:
```

```
            return eg_index-1
        else:
            return eg_index
```

K个最接近的元素

在函数find_closest_index基础上继续思考，已经获得了closest_index， 只需要找到剩余的K-1个元素就可以了。

剩余的K-1个元素，一定分布在closest_index 的两侧，借鉴归并排序的思路，把closest_index 两侧的元素视为两个数组，分别从left_index(closest_index-1)和right_index(closest_index + 1)开始进行遍历比较，如果lst[left_index] <= lst[right_index]则lst[left_index] 就是要找的那K-1个元素里的一个，此时left_index -= 1，游标向左滑动，继续进行比较

这里要考虑边界情况，如果left_index 超出了数组的索引范围，该如何比较呢，一侧超出了索引范围，剩余的最近元素一定集中在另一侧，此时，可以直接从另一侧数组里直接取剩余数量的元素，但我认为这样程序就会写的很复杂，要判断两侧的索引是否超出范围，要写很多if语句。

不弱这样处理，如果索引超出了范围，就认为这个索引上的元素是无穷大，这样就不需要判断左右两个索引是否超出范围以及取另一侧的剩余最近元素了。

示例代码

```
def find_closet_ele(lst, target, k):
    closet_ele_lst = []
    closest_index = find_closest_index(lst, target)   # 距离target最近的元素的位置
    closet_ele_lst.append(lst[closest_index])

    ele_count = 1
    left_index = closest_index - 1
    right_index = closest_index + 1
    # 借鉴归并排序思路,向两侧滑动遍历
    while ele_count < k:
        left_ele = get_ele(lst, left_index)
        right_ele = get_ele(lst, right_index)
        # 哪边元素距离target更近,哪边就走一步
        if target - left_ele <= right_ele - target:
            closet_ele_lst.append(left_ele)
            left_index -= 1
        else:
            closet_ele_lst.append(right_ele)
```

```
            right_index += 1
        ele_count += 1

    closet_ele_lst.sort()
    return closet_ele_lst


def get_ele(lst, index):
    # 索引超出范围的,返回无穷大
    if index < 0 or index >= len(lst):
        return float("inf")
    return lst[index]
```

完整代码

```
def find_closet_ele(lst, target, k):
    closet_ele_lst = []
    closest_index = find_closest_index(lst, target)   # 距离target最近的元素的位置
    closet_ele_lst.append(lst[closest_index])

    ele_count = 1
    left_index = closest_index - 1
    right_index = closest_index + 1
    # 借鉴归并排序思路,向两侧滑动遍历
    while ele_count < k:
        left_ele = get_ele(lst, left_index)
        right_ele = get_ele(lst, right_index)
        # 哪边元素距离target更近,哪边就走一步,这里注意取绝对值
        if abs(target - left_ele) <= abs(right_ele - target):
            closet_ele_lst.append(left_ele)
            left_index -= 1
        else:
            closet_ele_lst.append(right_ele)
            right_index += 1
        ele_count += 1

    closet_ele_lst.sort()
    return closet_ele_lst


def get_ele(lst, index):
    # 索引超出范围的,返回无穷大
    if index < 0 or index >= len(lst):
        return float("inf")
```

```
    return lst[index]


def find_closest_index(lst, target):
    """
    寻找距离target最近的位置
    :param lst:
    :param target:
    :return:
    """
    eg_index = find_eg_index(lst, 0, len(lst)-1, target)
    if eg_index == len(lst):
        return eg_index - 1
    elif eg_index == 0:
        return 0
    else:
        if target - lst[eg_index-1] <= lst[eg_index] - target:
            return eg_index-1
        else:
            return eg_index


def find_eg_index(lst, start, end, target):
    """
    找到第一个大于等于target的元素的位置，如果lst中最大的元素小于target
    则返回len(lst)
    :param lst:
    :param start:
    :param end:
    :param target:
    :return:
    """
    if start > end:
        return end + 1
    middle = (start + end)//2
    middle_value = lst[middle]
    if middle_value == target:
        return middle
    elif middle_value < target:
        return find_eg_index(lst, middle+1, end, target)
    else:
        return find_eg_index(lst, start, middle-1, target)


if __name__ == '__main__':
    lst = [1, 5, 8, 9, 17, 20]
```

```
    print(find_closet_ele(lst, 10, 4))
```

8.16 实现全排列算法

题目要求

给定数组，求数组的全排列
例如下面的数组

```
[1, 2, 3, 4]
```

它的全排列有24种：

```
[1, 2, 3, 4]
[1, 2, 4, 3]
...
[4, 1, 3, 2]
[4, 1, 2, 3]
```

思路分析

1、中学数学

这种排列组合有多少种，在中学时代就学过，4*3*2*1 = 24种排列组合。

第一位有4个选择，第一位确定后，第二位有3个选择，同理，第三位有2个选择，最后一位只有一个选择。

当我们用程序实现时，想要的不是排列组合的数量，而是要输出这些组合的内容，就不仅仅是一个算出一个数值那么简单。但算法归根结底还是数学问题，因此，我们可以回忆借鉴数学课上所学的方法，来考虑程序的算法逻辑。

2、递归算法

第一步，确定第一个位置的情况，你可以写一个循环，让每个元素都有一次机会放在第一个位置上

第二步，对于剩余的三个元素，问题转变成这3个元素有多少全排列情况，这就变成了递归

经过上面两步思考，我们大致有了这样的代码逻辑

```
def perm(lst, start, end):
    for index in range(start, end):
        lst[index], lst[start] = lst[start], lst[index]
        perm(lst, start+1, end, com_all_lst)

lst = [1, 2, 3, 4]
perm(lst, 0, len(lst))
```

列表中每个元素都有机会成为第一个元素，剩余的元素递归做同样的事情。

3、终止条件

递归必须有终止条件，而且这个终止条件非常容易找到，start 和 end相等时，就表明没有什么元素需要经过变换位置来组成新的排列组合了。

不过到目前位置，我们还没有把这些排列组合记录下来，于是，有了下面的改进

```
import copy

def perm(lst, start, end, com_all_lst):
    if start == end:
        com_all_lst.append(copy.deepcopy(lst))
    else:
        for index in range(start, end):
            lst[index], lst[start] = lst[start], lst[index]
            perm(lst, start+1, end, com_all_lst)

lst = [1, 2, 3, 4]
com_all_lst = []
perm(lst, 0, len(lst), com_all_lst)
print(com_all_lst)
```

程序运行结果

```
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3],
[1, 4, 3, 2], [1, 2, 3, 4], [1, 2, 4, 3],
[2, 1, 4, 3], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1],
[2, 1, 4, 3], [2, 1, 3, 4], [3, 1, 2, 4], [3, 1, 4, 2],
[3, 4, 1, 2], [3, 4, 2, 1], [3, 1, 2, 4], [3, 1, 4, 2], [2, 1, 4, 3],
[2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1], [2, 1, 4, 3], [2, 1, 3, 4]]
```

竟然出现了一些重复的排列组合，看来程序的逻辑存在bug。

4、还原列表

问题出在这行代码上

```
lst[index], lst[start] = lst[start], lst[index]
```

每一次递归调用时，我们将元素的顺序进行了调整，注意看 [3, 1, 4, 2] 这个结果，这是3放到第一个位置时的组合之一，此时列表最后一个元素是2，还记得全排列的思路么，让每一个元素都有机会放在第一个位置上，这个逻辑对应到代码

```
for index in range(start, end):
```

期初，列表里的元素是 1 2 3 4 ，这个循环希望将1 2 3 4 都放在第一个位置上一次，但是当 [3, 1, 4, 2] 出现时，元素的位置已经发生变化，当index = 3 时， lst[index] = 2,这就导致了以2开头的全排列又被计算了一次，所以导致重复。

为了避免重复，我们需要将列表里的元素还原，每次递归调用结束后，将之前互换位置的元素再次互换位置，只需增加一行代码就可以

```
for index in range(start, end):
    lst[index], lst[start] = lst[start], lst[index]
    perm(lst, start+1, end, com_all_lst)
    lst[index], lst[start] = lst[start], lst[index]
```

示例代码

```
import copy

def perm(lst, start, end, com_all_lst):
    if start == end:
        com_all_lst.append(copy.deepcopy(lst))
    else:
        for index in range(start, end):
            lst[index], lst[start] = lst[start], lst[index]
            perm(lst, start+1, end, com_all_lst)
            lst[index], lst[start] = lst[start], lst[index]

lst = [1, 2, 3, 4]
com_all_lst = []
perm(lst, 0, len(lst), com_all_lst)
print(com_all_lst)
```

8.17 还原ip地址

题目要求

一个字符串，里面存储的是ip地址，但是.（点）被删除了，只剩下数字，请编写算法，还原这个ip地址，如果可以还原成多个，请逐一列出

```
输入: 192168119130
输出: ['192.168.119.130']

输入: 101224010
输出:['10.12.240.10', '10.122.40.10', '10.122.401.0',
      '101.2.240.10', '101.22.40.10', '101.22.401.0',
      '101.224.0.10']

输入: 101226010
输出: ['10.122.60.10', '101.22.60.10', '101.226.0.10']
```

思路分析

一个ip地址有四段，每段的范围是0~255，可以分四次从输入字符串中截取字符串，第一次截取时，必然是从字符串索引为0的位置开始，那么可以截取的长度是1， 2， 3。 这一段截取后，进行第二次截取，第二次截取的开始位置取决于第一次截取结束的位置，如果第一次截取的长度是2，那么第二次就要从索引为2的位置上开始截取，可以截取的长度是1， 2， 3， 第三次和第四次做同样的操作，这么一分析，你就应该想到用递归函数来做了。

需要考虑的情况是：

- 截取后的每一段，如果是以0开头，那么这一段只能是0，总不能出现10.21.01.23的ip地址吧，01是什么鬼
- 截取后的每一段，其值大小不能超过255
- 如果截取后剩余的部分长度超过剩余几段理论上的最大长度,那么这个截取就是不合理的，比如192168119130， 第一段只截取1，那么剩余部分92168119130 的长度是11，后面三段最长也就是9，所以这样截取是不合理的
- 如果截取后余下的部分长度是0，也是不合理的
- 如果截取后余下的部分长度比余下几段理论上的最小长度还小，也是不合理的，比如截取后余下部分长度是2，可是还有3段需要截取，这样就是不合理的

示例代码

```
# coding=utf-8


def restore_ip(ip):
    retsection_lst = restore_ip_ex(ip, len(ip), 0, 4)
    return ['.'.join(item) for item in retsection_lst]


def restore_ip_ex(ip, ip_len, start_index, k):
    if k == 0:
        return [[]]

    if ip_len > 20 or ip_len < 4:
        return []

    section_lst = []
    for i in range(1, 4):
        # 如果截取后剩余的部分长度超过剩余几段理论上的最大长度,那么这个截取就是不合理的
        # 如果剩余长度为0也是不合理的
        rest_len = ip_len - i - start_index
        if (rest_len == 0 and k != 1) or rest_len > (k-1)*3 or rest_len < k-1:
            continue

        section = ip[start_index:start_index+i]
        # 如果某段以0开头,那么这一段就是只能是0,否则就会出现1.012.240.10这种情况,012显然是不合理的
        if section[0] == '0' and len(section) != 1:
            continue

        if int(section) > 255:
            continue

        # 截取下一段
        res_lst = restore_ip_ex(ip, ip_len, i+start_index, k-1)
        for item in res_lst:
            item.insert(0, section)

        section_lst.extend(res_lst)

    return section_lst


if __name__ == '__main__':
```

```
    print restore_ip("192168119130")
    print restore_ip("101224010")
    print restore_ip("1111")
    print restore_ip("101226010")
```

8.18亲密字符串

题目要求

有两个字符串，A和B，如果A字符串内部交换任意两个位置的字符就能够让A和B相等，就认为两个字符串是亲密字符串。

```
示例1
输入：A="cccb" ,B="cccb"
输出：True
解释：任意两个c交换位置，都可以让A=B,他们是亲密字符串

示例2
输入：A="ccbcab", B="cccbab"
输出：True
解释：A内部交换A[2]和A[3]后，A=B，他们是亲密字符串

示例3
输入：A="ab", B="ab"
输出：False
解释：A字符串内部不论怎样交换，都不可能与B相同
```

思路分析

如果A==B成立，那么就必须满足，A中有某个字符出现了两次或以上，只有这样，才能满足交换两个字符后与B相等，这是示例1的情况

如果A!=B，满足亲密字符串的条件就是A与B有两处不相同，而且交换这两处不相同的字符后A==B成立，这是示例2 的情况

如果A B两个字符串的长度不相等，必然不是亲密字符串。

遍历两个字符串，统计相同索引字符不相同的次数，如果次数是0，那么就必须满足A字符串有某个字符出现了两次或以上，可以用一个集合来保存A中的字符，如过有某个字符出现了两次或以上，那么最终，集合的大小一定小于A字符串的长度。

如果不相同的次数是2，那么交错对比这两处字符，如果不相同次数既不是0也不是2，则一定

不是亲密字符串

示例代码

```
# coding=utf-8


def is_close_str(str_1, str_2):
    # 长度要一致,且不能为0
    if len(str_1) != len(str_2) or len(str_1) < 2:
        return False

    index = 0
    diff_index_lst = []             # 记录不一致的索引位置
    char_set = set()
    while index < len(str_1):
        # 遇到相同位置字符不相同的情况
        if str_1[index] != str_2[index]:
            diff_index_lst.append(index)
            if len(diff_index_lst) > 2:
                return False
        char_set.add(str_1[index])
        index += 1

    if len(diff_index_lst) == 0:
        if len(str_1) != len(char_set):
            return True
    elif len(diff_index_lst) == 2:
        diff_index_1 = diff_index_lst[0]
        diff_index_2 = diff_index_lst[1]
        # 两处不一样的地方交错比较
        if str_1[diff_index_1] == str_2[diff_index_2] and \
                        str_1[diff_index_2] == str_2[diff_index_1]:
            return True

    return False

if __name__ == '__main__':
    print(is_close_str("", ""))
    print(is_close_str('abc', 'acba'))
    print(is_close_str('ab', 'ab'))
    print(is_close_str('cccb', 'cccb'))
    print(is_close_str('ccbcab', 'cccbab'))
    print(is_close_str('acbcad', 'dcbcaa'))
```

8.19 最大正方形

题目要求

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积

```
输入:

1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0

输出: 4
```

思路分析

只包含1的正方形可以出现在矩阵的任意位置，所以，不要妄想通过什么巧妙的办法快速的找到这个正方形的位置，正确的做法是遍历整个矩阵，尝试去找正方形，可能会找到多个，因此还要保留记录那个最大的正方形。

既然是遍历，那么就得从头到尾的进行遍历，假设矩阵matrix大小是M*N，那么就从matrix[0][0]开始进行遍历，一直遍历到matrix[M−1][N−1]。

当遍历到matrix[x][y]时，可以将这个点视为正方形的左上角，以此为基础，去判断matrix[x+1][y]， matrix[x][y+1]， matrix[x+1][y+1]这3个点是否也是1，如果成立，就找到了一个正方形，在此基础上，再向右下方扩展一层，判断能否以matrix[x][y]做正方形的左上角，以matrix[x+2][y+2]做正方形的右下角。

由于总是以某个点做正方形的左上角，因此，最后一行和最后一列是不需要遍历的，因为这些点无法构成正方形的左上角。

示例代码

```
matrix = [
    [1, 0, 1, 0, 0],
    [1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1],
    [1, 0, 1, 1, 1],
    [1, 0, 1, 1, 1],
```

```
]

matrix = [['1']]

def max_square(matrix):
    y = len(matrix)
    if y == 0:
        return 0

    x = len(matrix[0])
    if x == 0:
        return 0

    max_area = 0
    for i in range(0, y):
        for j in range(0, x):
            cell = matrix[i][j]
            if cell == '0':
                continue

            area = get_area(matrix, i, j)
            if area > max_area:
                max_area = area

    return max_area


def get_area(matrix, left_up_x, left_up_y):
    step = 1
    right_down_x = left_up_x + step
    right_down_y = left_up_y + step

    while right_down_x < len(matrix) and right_down_y < len(matrix[0]):
        b_square = True
        if matrix[right_down_x][right_down_y] == '0':
            break

        for i in range(left_up_x, right_down_x):
            if matrix[i][right_down_y] == '0':
                b_square = False
                break

        if not b_square:
            break

        for j in range(left_up_y, right_down_y):
```

```
            if matrix[right_down_x][j] == '0':
                b_square = False
                break

        if not b_square:
            break

        step += 1
        right_down_x = left_up_x + step
        right_down_y = left_up_y + step


    return step**2


if __name__ == '__main__':
    print(max_square(matrix))
```

8.20接雨水

题目要求

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水

上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）

思路分析

思考才是最重要的

这个题目非常好的诠释了一句话----“技术，只是实现目标的方式和手段”。

很多人面对问题时，总是有一种较劲脑汁写算法的冲动，还有人，以为每一种问题都有一个成熟的解决方案，似乎接触过这个方案并记忆下来，今后再遇到时就可以迎刃而解了。殊不知，编程是一个创造性的思考过程，很多人不愿意思考，更喜欢走马观花般的背答案。短时间内的确可以提升算法能力，但长期来看，你的逻辑能力才是解决问题的根本，思考才是第一位的。

两个基本判断

我们先不去想怎么写算法，而是先分析，抽丝剥茧的对这个问题进行分析。

首先，最左侧的柱子和最右侧的柱子上面不可能存储雨水，这个判断很简单，不做讨论。

其次，每个柱子上能存储多少水，取决于它左右两侧最高柱子的高度，以索引为5的柱子为例，这个柱子的高度是0，它左侧最高的柱子是索引为3的柱子，高度为2， 它右侧最高的柱子是索引为7的柱子，高度为3，这两个柱子的高度决定了，索引为5的柱子上面可以存储水2个单位。

算法思路

经过上面的两点分析，解题的思路就非常清晰了。

遍历数组，计算每个柱子左右两侧的最高柱子的高度，这两个高度中选最小的，如果这个最小高度大于当前柱子的高度，那么差值就是当前这个柱子能够存储的水的量。

由于遍历的过程是从左向右，因此，左侧最高柱子的高度可以用一个变量保存起来，而右侧最高柱子的高度则需要每次都去计算,但是经过一次寻找后，这个最高柱子的位置就确定了，假设位置为K，对于这个柱子左侧的所有柱子来说，他们右侧的最高柱子都是固定的，因此，也可以存储起来，直到遍历过程中当前柱子的索引大于等于K，则需要重新寻找。

实例代码

```
def trap(lst):
    left_height = 0
    right_index = 0
    rain = 0
    for index, item in enumerate(lst):
        # 第一个和最后一个不用考虑，因为柱子上方无法存储水
        if index == len(lst) - 1 or index == 0:
            left_height = item
            continue

        # 如果左侧最高为0或者和当前高度相同，当前这个柱子的上方无法存储水
        if left_height == 0 or left_height == item:
            left_height = item
```

```
            continue

        # 获取右侧最高的柱子高度，  取左右两侧较矮的，如果当前柱子高度小于它，则可以存储水
        if index >= right_index:
            right_index, right_heigth = find_right_heigth(index, lst)

        min_heigth = min([left_height, right_heigth])
        if min_heigth > item:
            rain += min_heigth - item

        # 更新左侧最高柱子的高度
        if item > left_height:
            left_height = item

    return rain


def find_right_heigth(index, lst):
    """
    返回index右侧最高柱子的位置和高度
    :param index:
    :param lst:
    :return:
    """
    right_heigth = 0
    right_index = index
    for i in range(index, len(lst)):
        if lst[i] > right_heigth:
            right_index = i
            right_heigth = lst[i]

    return right_index, right_heigth


if __name__ == '__main__':
    lst = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

    print(trap(lst))
```

8.21 字符串的排列

题目要求

给定两个字符串 s1 和 s2，写一个函数来判断 s2 是否包含 s1 的排列。

换句话说，第一个字符串的排列之一是第二个字符串的子串

实例1

```
输入: s1 = "ab" s2 = "eidbaooo"
输出: True
解释: s2 包含 s1 的排列之一 ("ba")
```

实例2

```
输入: s1= "ab" s2 = "eidboaoo"
输出: False
```

思路分析

暴力算法

以字符串“abc”为例，它有6种排列，分别为

- abc
- acb
- bac
- bca
- cab
- cba

你可以写一个算法，找出一个字符串的所有排列，然后利用字符串的find函数，在第二个字符串中逐个查找是否存在这些排列。

上面的思路虽然笨重，但可以解决问题。

更优的解法

让我们把思路从字符串排列上移开，假如有一个字符串包含了字符串“abc”的6个人排列中的某一个，那么可以肯定的讲，在这个字符串里，一定有一个长度为3的子串，这个子串包含了a,b,c这三个字符。

至于顺序，则完全不用考虑，只要这个长度为3的子串包含了a,b,c这三个字符，那么这个子串就一定是字符串“abc”的某个排列。

算法思路如下

1. 统计第一个字符串的字符信息，存放于字典中，记录每个字符的数量
2. 遍历第二个字符串，同样统计每个字符的数量，然后和第一步中的字符信息进行比较，其实就是两个字典的比较。这里有一个需要注意的地方，遍历过程中，只能统计子串的信息，而不是从头到尾的去做统计，如果第一个字符串的长度是3，那么当第二个字符串遍历到索引5时，只能统计索引3到5这个范围内的信息

   如果两个字典完全相同，就说明第二个字符串的这段子串是第一个字符串的一个排列

实例代码

```
def check_inclusion(s1, s2):
    s1_chr_info = {}
    for chr in s1:
        s1_chr_info.setdefault(chr, 0)
        s1_chr_info[chr] += 1

    s2_chr_info = {}
    start_index = 0     # 连续空间开始的位置
    for index, chr in enumerate(s2):
        if chr not in s1_chr_info:
            s2_chr_info.clear()
            start_index = index+1
            continue
        # 右侧进
        s2_chr_info.setdefault(chr, 0)
        s2_chr_info[chr] += 1

        # 左侧出
        left_index = index - len(s1)
        if left_index >= start_index and s2[left_index] in s1_chr_info \
                and s2[left_index] in s2_chr_info :
            s2_chr_info[s2[left_index]] -= 1

        if is_dict_same(s1_chr_info, s2_chr_info):
            return True

    return False


def is_dict_same(dict_1, dict_2):
    """
    dict_1 是第一个字符串的统计信息，以它为基准进行判断
    :param dict_1:
    :param dict_2:
```

```
    :return:
    """
    if len(dict_1) != len(dict_2):
        return False

    for k, v in dict_1.items():
        if k not in dict_2:
            return False
        if v != dict_2[k]:
            return False

    return True


if __name__ == '__main__':
    print(check_inclusion("ab", "eidbaooo"))
    print(check_inclusion("abcd", "eidbadcooo"))
    print(check_inclusion("abcd", "eicwdbardcooo"))
```