深 圳 大 学 实 验 报 告

课程名称：Python 程序设计

实验项目名称：实验 3：数据结构

学院：计算机与软件学院

专业：软件工程

指导教师：潘浩源

报告人：郑彦薇 学号：2020151022

实验时间：2022/03/25~2022/4/5

实验报告提交时间：2022/04/05

教务部制

一、实验目的：

熟悉对 python 中列表、字典、元组等数据结构的操作

二、实验方法步骤

1、读题，对每个问题提出解决问题的思路

2、对需要编程解决的题目按照得到的思路编写源代码

三、实验过程及内容：

（一）列表的理解：

1、解决问题的思路与方法：

测试以下三种情况，解释输出结果的原因

（1）s = [0] * 3

```
print(s)
s[0] += 1
print(s)
```

对于上述代码，测试得到代码运行结果为：

```
[0, 0, 0]
[1, 0, 0]
```

原因解释：

s=[0]*3 创建了一个长度为 3，元素全为 0 的列表 s，对 s 进行输出，得到第一行结果；

s[0]+=1 对已创建列表中索引为 0 的元素即“0”进行加一的操作得到 1，再对 s 进行输出，得到第二行结果。

（2）s = [''] * 3

```
print(s)
s[0] += 'a'
print(s)
```

对于上述代码，测试得到代码运行结果为：

```
['', '', '']
['a', '', '']
```

原因解释：

s = [''] * 3 创建了一个长度为 3，元素全为空的列表 s，对 s 进行输出，得到第一行结果；

s[0] += 'a'将已创建列表中索引为 0 对应的元素进行加‘a’操作，第一个元素的值被更改为 a，对 s 进行输出，得到第二行结果。

（3）s = [[]] * 3

```
print(s)
s[0] += [1]
print(s)
```

对于上述代码，测试得到代码运行结果为：

```
[[], [], []]
[[1], [1], [1]]
```

原因解释：

s=[[]]*3 创建了一个长度为 3，元素全为“[]”的列表 s，对 s 进行输出，得到第一行结果；因为 s 的每个元素都为一个列表，当使用*运算符时，包含列表的列表重复创建新列表时，实质是对创建已有对象的引用，因此在对 s[0]进行 s[0]+=[1]的修改时，相应的引用也就是另外两个空列表“[]”也会被修改。故对 s 进行输出，得到第二行结果。

2、遇到的问题和收获：

在测试最后一段代码时，一开始对于出现的结果并不确定出于什么原因。在测试时插入 s.append([])

print(s)并更改修改项的索引为 2（如下图所示）：

```
1  s = [[]] * 3
2  print(s)
3  s.append([])
4  print(s)
5  s[2] += [1]
6  print(s)
```

得到的输出为：

```
[[], [], []]
[[], [], [], []]
[[1], [1], [1], []]
```

从而确定通过*运算符得到的前三个“[]”实质上属于同一个对象，后来插入的“[]”为另一个对象，也反过来验证原测试代码的结果的合理性。

（二）元组的理解：

1、解决问题的思路与方法：

（1）下面代码是否正确？解释原因

t = (1, 2, 3)

t.append(4)

t.remove(0)

t[0] = 1

以上的代码是不正确的。

原因解释：t=(1,2,3)创建的是元组，而 append、remove 都是列表的功能，在这里并不能正确使用。另外元素 0 并不在元组中，也不能正常进行删除。

（2）下面代码是否正确？解释原因

t1 = (1, 2, 3, 7, 1, 0, 5)

t2 = (1, 2, 5)

t1 = t2[:]

以上代码是不正确的。

原因解释：根据 t1,t2 元组的元素，首先 t1 为“（1，2，3，7，1，0，5）”对于元组有：mytuple[:]表示取元组 mytuple 中的所有元素，故 t2[:]的结果为“（1，2，5）”，很明显 t1 与 t2[:]并不相等。

（3）切片：测试下面代码，解释输出的结果

```
t = (1, 2, 2, 7, 8, -2, 5)
print(t[3])
print(t[1: 3])
print(t[-1:-3])
print(t[-1:-3:-1])
print(t[-1: : -3])
print(t[ : -1: 3])
print(t[3 : -1: 3])
print(t[3 : -1: -3])
```

对于上述代码，测试得到的运行结果为：

```
7
(2, 2)
()
(5, -2)
(5, 7, 1)
(1, 7)
(7,)
()
```

原因解释：

①print(t[3])：根据列表 t 的内容，该语句输出索引为 3 的元素，即输出元素 7；

②print(t[1:3])：对列表进行切片，start=1，end=3，输出列表中索引从 1~3−1 的元素，即输出片段（2，2）；

③print(t[−1:−3])：对列表进行切片，start=−1，end=−3，按 start 和 end 的值，目的是输出列表中索引从−1 的元素依次取到索引位−3 的前一位元素，在当前列表中表现为从右往左进行输出，但默认输出顺序为左到右，方向相反，故最终输出结果为()；

④print(t[−1:−3:−1])：在上述语句的基础上添加 step=−1，表示所取片段进行从右往左的输出，与默认 start 到 end 的顺序相同。根据 start=−1，end=−3，依次从索引为−1 的元素即 5 往左取到索引为−3 即 8 的“前一位”−2，故输出（5，−2）；

⑤print(t[−1::−3])：表示从索引−1 开始，从右往左，每 3 位取一个值，即从 5 开始往左，依次取出 7，1，最终输出（5，7，1）；

⑥print(t[:−1:3])：默认从列表的第一个元素开始，到索引为−1−1=−2 即−2 结束，每三位取一个值，依次取得 1，7，故输出（1，7）；

⑦print(t[3:−1:3])：因为 start=3，end=−1，step=3 表示从索引为 3 的元素开始依次往后取到索引为−1 的前一位即索引为−2 结束，每 3 位取一个值，取出元素 7。因为在元组中，若元组只包含一个元素，最后需多写一个逗号，故最终输出（7，）

⑧print(t[3:−1:−3])：因为 start=3，end=−1，step=−3 表示从索引为 3 的元素开始从右往左，每三位取一个值，直到索引为−1 的元素在当前顺序下的前一位结束，根据元组 t 可以知道，索引为−1 的元素为 5，索引为 3 的元素为 7，从 7 到 5 为左到右，而根据 step 规定，取

值顺序为从右到左，方向相反，故最终输出().

2、遇到的问题和收获：

通过对第三段代码的运行测试，根据结果对切片功能有了更深刻的认识。

（三）函数+列表：

1、解决问题的思路与方法：

写出每段代码的输出,解释原因

(a)
```
def f(n):
    n=5
m = 2
f(m)
print(m)
```

测试得到代码运行结果为:

```
2
```

原因解释:

传入函数 f(n)的 m 值在函数中被改为 5，但这个值并没有返回给主函数，也就是 m=5 这个结果只保留在 f(n)函数中，对 m 进行输出，得到的结果还是 2.

(b)
```
def f(L):
    L[0] = 42
    print(L[0])
L = [1,2,3]
f(L)
print(L[0])
```

测试得到代码运行结果为:

```
42
42
```

原因解释:

传入函数 f(L)的列表 L 与主函数中的 L 为同个对象，在函数中，将列表中索引为 0 的元素赋值为 0，即列表 L 中元素 L[0]更改为 42，故在函数 f(L)中以及在主函数中，对 L[0]进行输出，都可以得到结果 42.

(c)
```
def f():
    n = 5
n=4
n=f()
print(n)
```

测试得到代码运行结果为:

```
None
```

原因解释:

主函数中首先将 n 赋值为 4，又令 n=f()，进入函数 f()后，得到一个 n=5 的结果，但这个结果并没有返回给主函数，而是返回一个默认的值 None，而函数外的 n 一开始的值 4 又已被更改为 f()即 None，故最终输出结果 None。

(d) **def f(L):**
L2 = L
L = [1,2]
L[0] = 5
print(L)
L = [2,3]
print(L2)

测试得到代码运行结果为：此段代码不能正常输出
原因解释：在主函数中并没有定义 L2 变量，不能正常运行并输出。

2、遇到的问题和收获：

通过对代码(c)的测试，认识了在 python 中，就算无 return 语句，函数也会执行 return 的逻辑，且默认返回值为 None。

（四）字典的理解：

1、解决问题的思路与方法：

（1）以下哪些字典创建是有效的，哪些是无效的？解释原因。

d = {[1, 2]:1, [3, 4]:3}
d = {(1, 2):1, (3, 4):3}
d = {{1, 2}:1, {3, 4}:3}
d = {"12":1, "34":3}

以上字典创建中，d = {[1, 2]:1, [3, 4]:3}、d = {{1, 2}:1, {3, 4}:3}是无效的，d = {(1, 2):1, (3, 4):3}、d = {"12":1, "34":3}是有效的。

原因解释：

创建字典时，可以通过指定每个 key:value 对来初始化字典，但是对于字典的 key，要求 key 为任意不可变数据。而[1,2]、[3,4]以及{1,2}、{3,4}分别属于列表和集合或字典，都是可变数据类型，不能作为字典中的 key 使用。相反，(1,2)、(3,4)和"12"、 "34"分别属于元组和字符串，都是不可变数据类型，可作为字典中的 key 使用。因此 d = {[1, 2]:1, [3, 4]:3}、d = {{1, 2}:1, {3, 4}:3}是无效的，d = {(1, 2):1, (3, 4):3}、d = {"12":1, "34":3}是有效的。

（2）基于以下代码，回答问题

（a）根据题意编写得到的代码如下：

```
1  D = {"what":22,"are":11,"you":14,"doing":5,"next":9,"Saturday?":4}
2  sum = 0
3  for x in D.items():
4      sum = sum+D[x[0]]
5  print(sum)
```

代码运行结果展示：

```python
D = {"what":22,"are":11,"you":14,"doing":5,"next":9,"Saturday?":4}
sum = 0
for x in D.items():
    sum = sum+D[x[0]]
print(sum)
```

for x in D.items()

test3

```
C:\Users\4334\PycharmProjects\pythonProject_test3\venv\Scripts\python.exe C:/U
65

Process finished with exit code 0
```

原因解释：

根据字典的创建规则和访问规则，可知 D.items()可以返回字典中的 key−value 对列表。通过 for 循环，在每一趟循环里对 D.items()得到的键值列表进行访问，且以 x 进行记录；又根据键值列表可以知道，在 x 中索引为 0 对应键值对中的 key 值，即 x[0]=key。对于字典，又可以通过 key 对字典进行索引获得 value 的值，即 D[x[0]]=D[key]=value，因此通过 D[x[0]]便可在每一趟循环中，依次获得字典中的 value 值，通过循环对所有值进行相加存储于 sum 中，最后对 sum 进行输出，即可得到结果 65.

（b）根据题意编写得到的代码如下：

```python
D = {"what":22,"are":11,"you":14,"doing":5,"next":9,"Saturday?":4}
sum = 0
for x in D.items():
    sum = sum+x[1]
print(sum)
```

代码运行结果展示：

```python
D = {"what":22,"are":11,"you":14,"doing":5,"next":9,"Saturday?":4}
sum = 0
for x in D.items():
    sum = sum+x[1]
print(sum)
```

test3

```
C:\Users\4334\PycharmProjects\pythonProject_test3\venv\Scripts\python.exe C:/U
65

Process finished with exit code 0
```

原因解释：

同理于问题（a）中所提，字典中可以通过 items()获得键值对列表，通过 for 循环对每次获得的键值对列表存储于 x 中，又根据键值对列表信息可以知道，在 x 中索引为 1 对应键值对中的 value 值，即 x[1]=value。在循环中，每次获得字典中的每对键值对，并通过索引 1 访问键值对中的值，进行相加，将结果存储于 sum 中，最后对 sum 进行输出，也可以获得相同的结果为 65.

2、遇到的问题和收获：

解决问题（1）时，对可变和不可变数据类型的认识更加深刻；解决问题（2）时，通过两个子问题掌握了两种访问字典中 value 值的方法。

（五）列表元素的增加：

1、解决问题的思路与方法：

根据题目要求，在计时开始和计时结束之间编写列表元素增加代码，统计不同方法下的运行时间差，采取步骤如下：

① 设置多层循环，重复增加元素过程，实现对短时间的统计

② 创建列表并赋初始值

③ 分别使用“+”和 append()增加列表元素，统计时间，实验中选择重复在列表中增加元素 10 进行对时间的统计

④ 输出时间统计结果，进行对比

（1）使用“+”增加元素：

首先根据题意，编写使用“+”增加元素的代码如下：

```
import time
myList = list(range(1, 10))
list1 = [10]
start = time.time()
t = 100000
while t != 0:
    myList = myList + list1
    t -= 1
print(time.time()-start)
```

重复列表增加元素过程 t 次

得到元素为 10 的列表 list1，加到 myList 中

运行结果如下：

```
15.349759817123413

Process finished with exit code 0
```

（2）使用 append()增加元素：

首先根据题意，编写使用 append()增加元素的代码如下：

```
import time
myList = list(range(1, 10))
start = time.time()
t = 10000000
while t != 0:
    myList.append(10)
    t -= 1
print(time.time()-start)
```

重复列表增加元素过程 t 次

增加元素 10

运行结果如下：

```
1.294179916381836

Process finished with exit code 0
```

2、遇到的问题和收获：

（1）**实验发现**：通过上述运行结果，可以知道，使用“+”增加列表元素时，重复 t=100000 次的实际运行时间约为 15.35，而使用 append()函数增加列表元素时，重复 t=10000000 次的时间也仅仅约为 1.29，可以知道使用 append()增加列表元素的效率时远远高于使用“+”增加列表元素的效率。

（2）实验收获：通过该实验学习 python 环境中对程序运行时间的统计，同时也认识了不同的增加元素的方法的运行效率差异，可以更好的选择，增加元素的方式。使用“+”增加列表元素时，需要创建新列表，这一过程也会增加内存的消耗，综合效率与内存，在增加列表元素时，应尽可能的选择 append()方法。

（六）合并两个排序的列表：

1、解决问题的思路与方法：

（1）不使用 sort()或 sorted()：

Merge 函数思路和方法：

① 创建一个新的空列表为传入的两个列表相加，获取传入的两个列表的长度

② 通过下标索引比较当前较小值，因为传入函数的两个列表已经有序，因此只需按照列表顺序依次对当前索引所指数据进行比较

③ 索引从 0 开始，进行比较，将较小值存入 list，并对该值所在列表的索引+1，进行下一个值的比较，直到已经完成 list1 或 list2 最后一个数的比较

④ 比较结束后，list1 或 list2 可能仍有未比较的数，设置循环将这些数依次存入 list 即可得到新的有序序列

具体编程如下：

```python
def merge(list1, list2):
    len1 = len(list1)
    len2 = len(list2)
    List = list1 + list2
    i = j = k = 0
    while i < len1 and j < len2:
        if list1[i] < list2[j]:
            List[k] = list1[i]
            i += 1
            k += 1
        else:
            List[k] = list2[j]
            j += 1
            k += 1
    while i < len1:
        List[k] = list1[i]
        k += 1
        i += 1
    while j < len2:
        List[k] = list2[j]
        k += 1
        j += 1
    return List
```

初始化 List 为传入的两个列表相加

对传入函数的 list1 和 list2 的元素进行比较，将当前的较小值存入 list 中，并将对应的下标加 1

```
l1 = input()
list1 = [int(n) for n in l1.split()]
List1 = list(list1)
l2 = input()
list2 = [int(n) for n in l2.split()]
List2 = list(list2)
print(merge(List1,List2))
```

将被默认为字符串的输入转换为列表，使程序能够正常运行

运行结果如下：

输入：[2，4，7]，[1，5，6]

```
2 4 7
1 5 6
[1, 2, 4, 5, 6, 7]
```

（2）使用 sort()或 sorted()：

思路和方法：

① 输入两个列表元素，并将元素传入函数中

② 创建新列表为传入的两个列表相加

③ 调用 sort 函数进行排序

具体编程如下：

```
def merge(list1, list2):
    List = list1 + list2
    List.sort()
    return List
l1 = input()
list1 = [int(n) for n in l1.split()]
List1 = list(list1)
l2 = input()
list2 = [int(n) for n in l2.split()]
List2 = list(list2)
print(merge(List1,List2))
```

调用 sort()进行排序

将被默认为字符串的输入转换为列表，且元素为整型

运行结果如下：

输入：[2，4，7]，[1，5，6]

```
2 4 7
1 5 6
[1, 2, 4, 5, 6, 7]
```

2、遇到的问题与收获：

以使用 sort()函数排序得到的结果举例，在一开始进行输入时，只是简单的将输入转换为列表，得到的结果如下所示：

```
2 4 7
1 5 6
['1', ' ', '2', ' ', '4', ' ', '5', ' ', '6', '7']
```

这是因为输入的“2 4 7”中，空格和数字都被默认为字符串，在输出时也是以字符形式进行输出，且空格会被输出。因此在具体编程中应先将输入转换为整型数组，再将其转换为列表进行操作，从而得到正确结果。

（七）子列表：

1、解决问题的思路和方法：

① 根据传入的列表获得列表的长度

② 按照长度设置循环，首先判断 list2 的第一个元素是否在 list1 中：从 list1 第一个元素开始，依次往后判断 list1 中是否存在与 list2 第一个元素相同的值

③ 若不在，说明 list2 一定不是 list1 的子序列，返回 False

④ 若在，则按照子序列的特点，从 list2 的第一位及其在 list1 中的位置开始，依次往后判断 list1 和 list2 的元素是否一一对应，若不对应，返回 False；若 list2 元素已经全部遍历，则返回 True。

具体编程如下：

```
def match_pattern(list1, list2):
    len1 = len(list1)
    len2 = len(list2)
    i = j = 0
    while j < len1:
        if list2[i] == list1[j]:
            i += 1
            j += 1
            while i < len2 and j < len1:
                if list2[i] != list1[j]:
                    return False
                i += 1
                j += 1
            if i == len2:
                return True
        j += 1
    return False

l1 = input()
list1 = [int(n) for n in l1.split()]
List1 = list(list1)
l2 = input()
list2 = [int(n) for n in l2.split()]
List2 = list(list2)
if match_pattern(List1,List2):
    print("True")
else:
    print("False")
```

首先判断 list2 的第一个元素是否在 list1 中，若存在，则进入进一步判断的循环

对第一个元素之后的元素依次进行判断，若不能够一一对应，说明 list2 不为 list1 的子列表，返回 False

若 list2 的所有元素已经遍历，且不存在上述不成立的情况，则 list2 为 list1 的子列表，返回 True

对输入的强制转换

运行结果如下：

List1 统一设定为[4，10，2，3，50，100]

输入 list2 为[3，2，50]：

```
4 10 2 3 50 100
3 2 50
False

Process finished with exit code 0
```

输入 list2 为[2，3，50]：

```
4 10 2 3 50 100
2 3 50
True

Process finished with exit code 0
```

输入 list2 为[2，3，40]：

```
4 10 2 3 50 100
2 3 40
False

Process finished with exit code 0
```

2、遇到的问题与收获：

实验进行的是子列表的判断，需注意子列表除了满足每一个元素都在主列表中，还要注意顺序以及连续性。

（八）分饼干

1、解决问题的思路和方法：

① 首先对贪心算法的思路进行简述：贪心算法是从问题的初始状态出发，通过若干次贪心选择得到最优解，即对当前的情况做出最好的选择以得到最优解或较好的解，在此题的应用为：用小的饼干满足小的胃口

② 根据贪心算法，首先对获得的贪吃指数和饼干尺寸列表进行从小到大的排序

③ 从两个列表的第一个元素开始，按照饼干尺寸的顺序，判断当前饼干尺寸能否满足当前胃口，若能，则得到满足的孩子数量加一，并以下一块饼干对下一个小孩的胃口进行比较和判断；若不能，则以下一块饼干的尺寸判断能否满足当前胃口

④ 循环上述过程直到遍历所有饼干尺寸或遍历所有小孩的贪吃指数，最后返回满足的孩子数量

具体编程如下：

```python
def Divide_cookies(list1,list2):
    list1.sort() #对贪吃指数排序
    list2.sort() #对饼干尺寸排序
    i = count = 0
    for j in list2: #遍历每一块饼干
        if j ≥ list1[i]: #如果饼干大于贪吃指数
            count += 1 #得到满足的孩子个数+1
            i += 1 #换下一个小朋友进行判断
        if i ≥ len(list1):
            break; #如果每一个小朋友都遍历，则结束分饼干
    return count
l1 = input()
list1 = [int(n) for n in l1.split()]
List1 = list(list1)
l2 = input()
list2 = [int(n) for n in l2.split()]
List2 = list(list2)
print(Divide_cookies(List1,List2))
```

运行结果如下：

输入贪吃指数：1，2，3；饼干尺寸：1，1：

```
1 2 3
1 1
1

Process finished with exit code 0
```

输入贪吃指数：1，2；饼干尺寸：1，2，3：

```
1 2
1 2 3
2

Process finished with exit code 0
```

2、遇到的问题和收获：

了解了如何利用贪心算法解决问题。

（九）“几乎对称”列表

1、解决问题的思路和方法：

① 首先判断输入列表是否已经为对称列表，若不是，才进一步进行判断

② 根据输入的列表获得列表的长度，根据长度设置双层循环

③ 在循环中，对元素进行两两互换，并判断当前互换所得列表是否为对称列表

④ 若互换后为对称列表，则说明满足几乎对称的条件，返回 True

⑤ 若循环结束仍不存在这样的两个元素交换后使得列表对称，说明输入的列表不是几乎对称列表，返回 False

具体编程如下：

```python
def is_symmetric(list):
    if list == list[::-1]:
        return True
    return False
def is_almost_symmetric(lst):
    for i in range(0, len(lst)-1):
        for j in range(i+1, len(lst)):
            temp = lst[:]
            temp[i] = lst[j]
            temp[j] = lst[i] #两两交换进行判断
            if is_symmetric(temp):
                return True
    return False
l = input()
lst = [int(n) for n in l.split()]
List = list(lst)
if is_symmetric(List): #对输入的列表进行判断
    print("False")
else:
    if is_almost_symmetric(List):
        print("True")
    else:
        print("False")
```

利用切片功能实现列表的倒序，从而进行判断

需注意：temp=list 并不能实现复制而是指向同一个列表，因此同样利用切片操作进行复制

运行结果如下：

输入[1，2，1，2]:

```
1 2 1 2
True

Process finished with exit code 0
```

说明：交换 lst[0]和 lst[1]得到[2，1，1，2]为对称列表
输入[1，2，3，4，5，1，4，3，2，5]:

```
1 2 3 4 5 1 4 3 2 5
True

Process finished with exit code 0
```

说明：交换 lst[5]和 lst[9]得到[1，2，3，4，5，5，4，3，2，1]为对称列表
输入[1，2，3，3，2，0]:

```
1 2 3 3 2 0
False

Process finished with exit code 0
```

2、遇到的问题和收获：

在进行列表复制时，应注意 temp=list 只是让 temp 指针指向 list 所拥有的内容，并不能达到复制列表用于解决问题这一目的。

（十）矩阵

1、解决问题的思路和方法：

对于矩阵的输入：输入矩阵的行数 n，设置循环，输入每一行并转换为列表，添加为列表的新元素。

a. **Matrix_dim(M)：**

① 传入函数的列表的特点：可知列表中的每一个元素都为其所表示的矩阵中的每一行

② 根据上述特点，统计列表的元素个数即为矩阵的行数，每个元素（元素也为列表）的具体个数为矩阵的列数

③ 统计行数列数，创建另一个列表，将行、列数添加到新列表中，对新列表进行返回即可得到列表所表示矩阵的维度

具体编程如下：

```
def matrix_dim(M):
    lst = []
    lst.append(len(M))
    lst.append(len(M[0]))
    return lst
n = int(input())
M = []
for i in range(0,n):
    M.append(list(input().split(" ")))
print(matrix_dim(M))
```

获取行数和列数

将每一行输入转换为列表，作为新元素添加到列表 M 中

运行结果如下：

```
3
1 2
3 4
5 6
[3, 2]

Process finished with exit code 0
```

b. Mult_M_v(M,v)：

① 根据要求创建列表 v 且列表长度为 m 与矩阵 M 的列数相同，列表 v 表示向量，并将向量和矩阵传入函数中

② 按照矩阵与向量相乘的运算法则，访问矩阵中的行，将每一行中的元素与向量对应每行元素进行相乘相加

③ 根据矩阵和向量相乘的结果的特点，将每一行相乘结果作为新元素添加到新创建的列表 lst 中，且每一次得到的结果应先转换为列表，表示得到的结果为 n*1 矩阵

具体编程如下：

```
def mult_M_v(M,v):
    lst = []
    for i in range(0, len(M)):
        sum = 0
        for j in range(0,len(M[0])):
            sum += int(M[i][j])*int(v[j])
        lst.append([sum])
    return lst
n = int(input())
M = []
for i in range(0,n):
    M.append(list(input().split(" ")))
v = list(input().split(","))
print(mult_M_v(M,v))
```

访问矩阵每一行每一个元素，与向量相乘相加

每一个结果转换为列表，添加为 lst 的新元素

运行结果如下：

```
3
1 2
3 4
5 6
1,2
[[5], [11], [17]]

Process finished with exit code 0
```

c. Transpose(M)：

① 设置临时变量 temp，根据矩阵转置的特点，从矩阵的第一列开始，将列中的每一个元素存储到列表 temp 中

② 将每一列即 temp 作为新元素添加到 lst 中

③ 以矩阵 M 的列数设置循环，每一趟执行上述步骤，直到循环结束，返回 lst

具体编程如下：

```
def transpose(M):
    lst = []
    for i in range(0,len(M[0])):
        temp = []
        for j in range(0,len(M)):
            temp.append(int(M[j][i]))
        lst.append(temp)
    return lst
n = int(input())
M = []
for i in range(0, n):
    M.append(list(input().split(" ")))
print(transpose(M))
```

Temp 获取矩阵每一列元素

运行结果如下：

```
3
1 2
3 4
5 6
[[1, 3, 5], [2, 4, 6]]

Process finished with exit code 0
```

d. Largest_col_sum(M)：

① 根据矩阵的行数和列数设置循环，对每一列依次访问每一个元素，进行相加并将结果存储到 sum 中，sum 在获取新的一列元素和之前，应先重新赋值为 0

② Sum 与 max_sum 进行比较，若比 max_sum 大，则更新 max_sum 为 sum

③ 返回最大列元素和 max_sum

具体编程如下：

```python
def largest_col_sum(M):
    max_sum = 0
    for i in range(0,len(M[0])):
        sum = 0
        for j in range(0,len(M)):
            sum += int(M[j][i])
            if sum > max_sum:
                max_sum = sum
    return max_sum
n = int(input())
M = []
for i in range(0, n):
    M.append(list(input().split(" ")))
print(largest_col_sum(M))
```

传入函数的列表元素实质为 str 类型，在进行相加时需先转换为整型

运行结果如下：

```
2
5 6 7
0 -3 5
12

Process finished with exit code 0
```

e. Switch_columns(M,i,j)：

① 首先将传入函数的矩阵 M 的每一个元素转换为整型

② 根据 i，j，设置循环从第一行到最后一行，访问第 i 列和第 j 列的元素，进行交换

③ 循环结束，返回矩阵 M

具体编程如下：

```python
def switch_columns(M,i,j):
    for p in range(0,len(M)):
        for q in range(0,len(M[0])):
            M[p][q] = int(M[p][q])
    for k in range(0,len(M)):
        temp = M[k][i]
        M[k][i] = M[k][j]
        M[k][j] = temp
    return M
n = int(input())
M = []
for k in range(0, n):
    M.append(list(input().split(" ")))
i = int(input())
j = int(input())
print(switch_columns(M,i,j))
```

运行结果如下：

```
2
5 6 7
0 -3 5
0
1
[[6, 5, 7], [-3, 0, 5]]

Process finished with exit code 0
```

f. **Matrix.py:**

① 将上述 a~e 所得函数编写在同一个文件中并命名为 matrix.py

② 在另一个文件中进行调用，对同一个矩阵进行以上 5 种操作，输出相应的结果

具体编程如下：

matrix.py 文件中每个函数的编写与上述具体编程所示相同，这里不做重复的展示。

test_matrix.py 编程为：

```python
import matrix
n = int(input())
M = []
for i in range(0,n):
    M.append(list(input().split(" ")))
v = list(input().split(","))
i = int(input())
j = int(input())
print("矩阵的维度为：",matrix.matrix_dim(M)) # (a)
print("矩阵与向量v的乘积为：",matrix.mult_M_v(M,v)) # (b)
print("矩阵转置的结果为：",matrix.transpose(M)) # (c)
print("矩阵列元素总和最大为：",matrix.largest_col_sum(M)) # (d)
print("交换第i列和第j列得到矩阵：",matrix.switch_columns(M,i,j)) # (e)
```

运行结果如下：

```
2
5 6 7
0 -3 4
1,2,3
0
1
矩阵的维度为： [2, 3]
矩阵与向量v的乘积为： [[38], [6]]
矩阵转置的结果为： [[5, 0], [6, -3], [7, 4]]
矩阵列元素总和最大为： 11
交换第i列和第j列得到矩阵： [[6, 5, 7], [-3, 0, 4]]
```

2、遇到的问题和收获：

在 python 中，并不能实现矩阵的直接输入，需要通过列表的特点进行表示和转换。由于输入被默认为字符串，故在输入时要注意去除符号避免将空格或者逗号也被转换为列表元素。

（十一）统计关键字

1、解决问题的思路和方法：

① 首先将 nARQ.py 文件转换为 txt 文件，在 python 中打开，并将其存储在统计关键字程序 test3_11.py 所在文件夹中

② 将 txt 文本文档中的内容进行处理：首先将所有的符号转换为空格，使用 split()函数将单词以空格和换行为界进行划分，得到列表 wordTwo

③ 创建字典 dictword，其中所有关键字为 python 关键字，每一个关键字对应的值为关键字在文本中出现的次数，初始化为 0

④ 根据 words 设置循环，从第一个单词开始，若单词出现在字典中，则对应单词出现次数加一，即 value 值加一

⑤ 按照关键字顺序对各自在 nARQ 文本中出现的次数进行输出，并计算总次数

具体编程如下：

```
import string
#去除文本中的符号，将它们转换为空格
def removePunctuations(word):
    for ch in word:
        if ch in string.punctuation:
            word = word.replace(ch," ")
    return word
```

```
#以文本中的空格和换行符将单词分划出来，得到一个只存放了单词的列表
def getWords(filepath):
    file = open(filepath)
    wordOne = []
    while file:
        line = file.readline()
        word = line.split(' ')
        wordOne.extend(word)
        if not line:
            break
    wordTwo = []
    for i in wordOne:
        wordTwo.extend(i.split())
    return wordTwo
```

```python
#统计每个关键字在文本nARQ出现的次数，并计算总数
def getWordNum(words):
    dictword = {"and":0, "as":0, "assert":0, "break":0, "class":0,
                "continue":0, "def":0, "del":0, "elif":0, "else":0,
                "except":0, "False":0, "finally":0, "for":0, "from":0,
                "global":0, "if":0, "import":0, "in":0, "is":0, "lambda":0,
                "None":0, "nonlocal":0, "not":0, "or":0, "pass":0, "raise":0,
                "return":0, "True":0, "try":0, "while":0, "with":0, "yield":0}
    for i in words:
        if i in dictword:
            dictword[i] += 1
    for key, value in dictword.items():
        print(f'nARQ中{key}出现：{value}次')
    sum = 0
    for value in dictword.values():
        sum += value
    print(f'nARQ文件中关键字共出现：{sum}次')
```

```python

filepath = 'nARQ.txt'
file = removePunctuations(filepath)
words = getWords(filepath)
getWordNum(words)
```

运行结果如下：

```
nARQ中and出现：38次
nARQ中as出现：2次
nARQ中assert出现：0次
nARQ中break出现：4次
nARQ中class出现：6次
nARQ中continue出现：0次
nARQ中def出现：30次
nARQ中del出现：0次
nARQ中elif出现：26次
nARQ中else出现：4次
nARQ中except出现：0次
nARQ中False出现：8次
nARQ中finally出现：0次
nARQ中for出现：20次
nARQ中from出现：15次
nARQ中global出现：4次
nARQ中if出现：137次
nARQ中import出现：15次
```

```
nARQ中in出现：16次
nARQ中is出现：50次
nARQ中lambda出现：0次
nARQ中None出现：17次
nARQ中nonlocal出现：0次
nARQ中not出现：42次
nARQ中or出现：11次
nARQ中pass出现：6次
nARQ中raise出现：0次
nARQ中return出现：43次
nARQ中True出现：7次
nARQ中try出现：0次
nARQ中while出现：6次
nARQ中with出现：4次
nARQ中yield出现：0次
nARQ文件中关键字共出现：511次

Process finished with exit code 0
```

2、遇到的问题和收获：

学会了如何在 python 中对某个文本中单词数量的统计，在开始统计前如何对文本进行去标点符号、转换为一维列表等，如何利用字典元素特点统计关键字出现次数。在进行文本的访问时，将文本与.py 文件存放在同一个文件夹中可以直接调用。

四、实验总结：

1、该实验是利用 python 中列表、元组、字典等数据结构解决实际问题

2、在进行应用时，应注意不同数据结构不同的定义方式以及它们具备的不同功能

指导教师批阅意见：

成绩评定：

指导教师签字：

年　　月　　日

备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。