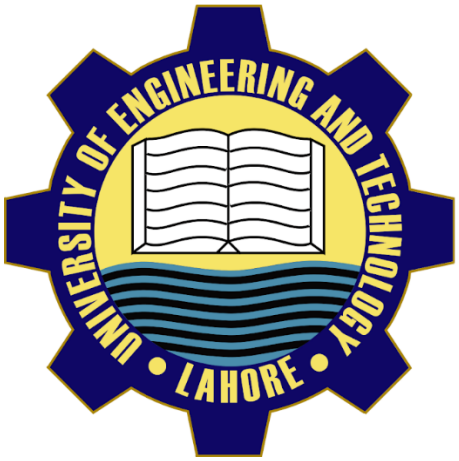
MCT-242 : COMPUTER PROGRAMMING-I

using Python 3.9



Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

https://www.youtube.com/playlist?list=PLWF9TXck7O_wMDB-VriREZ6EvwkWLNB7q

LAB 19: NESTED & TWO-DIMENSIONAL LIST: CLO-4

A list can contain elements of different data types. Likewise, it can even take another list as its element as shown here:

```
a=[5,3,4,[11,22,33],10]
```

The above list contains five elements out of which four are simple numbers and one is the list. The list element is at index **3** so if we want to access that we can do like:

```
print(a[3])
```

which will display following:

```
[11, 22, 33]
```

If we want to access the element of this nested list e.g., **22** is at index **1** of this nested list, we can do like:

```
print(a[3][1])
```

Convert a nested list into a flat list:

Suppose we have a nested list having few lists as its elements as shown here:

```
a=[[1,10],[5,11,22,32],[5,4,3]]
```

And we want to have a flat list having all elements of nested lists as single elements; nine single elements for above case. We can do it like:

```
a=[[1,10],[5,11,22,32],[5,4,3]]
flatA=[]
for i in a:
    for j in i:
        flatA.append(j)
print(flatA)
```

Now the **flatA** contains nine elements. A point of caution for above code is that if you have a single element in list **a** like:

```
a=[[1,10],[5,11,22,32],[5,4,3],100]
```

The above code will generate an error as the last element is not list but a simple integer and using **in** for single integer is not possible. However, we can resolve it by putting the single number as a list having single element as:

```
a=[[1,10],[5,11,22,32],[5,4,3],[100]]
```

And now the previous code will work perfectly.

Or even a better solution can be using a condition to check if the element is a list or not before the inner `for` loop.

```
a=[[1,10],[5,11,22,32],[5,4,3],100]
flatA=[]
for i in a:
    if(type(i)==list):
        for j in i:
            flatA.append(j)
    else:
        flatA.append(i)
print(flatA)
```

Two-Dimensional List

What if we want to define a 2-D matrix as:

$$A = \begin{bmatrix} -2 & 1 & 4 \\ 5 & 7 & 0 \\ 1 & 2 & 4 \end{bmatrix}$$

For such cases we can use Two-Dimensional lists. In fact, there is no difference at all in nested lists and 2-dimentional list. If we want to define the matrix shown above, we will do exactly the way we defined a nested list like:

```
A = [[-2,1,4],[5,7,0],[1,2,4]]
```

For our discussions in the course we will term 2-Dimentional list for the case when all nested lists contain same number of elements.

To display it like a matrix we can display it like:

```
print(A[0])
print(A[1])
print(A[2])
```

A much better way can be accessing individual elements and using proper format to display them as:

```
A = [[-2,1,4],[5,7,0],[1,2,4]]
for i in A:
    for j in i:
        print(f'{j:6.2f}',end=' ')
    print()
```

The format style `6.2f` means 6-digit width with 2 decimal places. The above code will display:

```
 -2.00   1.00   4.00
```

```
  5.00    7.00    0.00
  1.00    2.00    4.00
```

Accessing a particular row or column in a 2-D list:

Accessing one particular row is simple accessing the index of outer main list e.g. `A[1]` refers to nested list at index 1 i.e. the second row of the 2-D list. We can use that for any calculations also. For example, if we want to sum all elements of third row of 2-D list A, we can do it like:

```
A = [[-2,1,4],[5,7,0],[1,2,4]]
ans=sum(A[2])
print(ans)
```

Accessing a column is not direct as the row. One column e.g. first column is basically the index 0 of all rows. We will have to use loop to scan that index of all rows. For example, if we want to sum up all elements of column 2, it can be done as:

```
A = [[-2,1,4],[5,7,0],[1,2,4]]
sum=0
col=2
for i in range(len(A)):
    sum+=A[i][col]
print(sum)
```

If you need to access columns a bit often, you can also create your own function returning one column from the 2D list as simple 1D list and then you can use that for your calculations. The approach is shown here:

```
def column(A,c):
    a=[]
    for i in range(len(A)):
        a.append(A[i][c])
    return a

## Main Program ##
A = [[-2,1,4],[5,7,0],[1,2,4]]
col=column(A,2)
print(col)
print(sum(col))
```

Filling a 2D list with user input:

Suppose you want to fill a 5x5 list with user input, it can be done using nested loop. Outer loop will control the sublists and the inner loop will control the elements inside one sublist. See the code here:

```
A=[]
for r in range(5):
    print(f'Enter row-{r+1}.')
    a=[]
    for c in range(5):
        n=eval(input(f'Enter element-{c+1}:'))
        a.append(n)
    A.append(a)
print(A)
```

Tasks:

[1] Write a Python program to find the sublist in a list of sublists whose sum of elements is the highest. Do it by creating a user-defined function named as **`maxSum`** having one list of sublists as input argument and it should return the sublist with maximum sum.

An example case of Main Program is shown here:

```
a=[[2,3,1],[5,6,1,0,-2],[3,4],[5,6,7,2]]
print(maxSum(a))
```

It will have the output as:

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

[2] Write a program to split a given list into two parts where the length of the first part of the list is given.

[3] Write a Python program that will split a list every Nth element. Do it by creating a function **`splitList`**. The function will have one 1D list **`a`** and a number **`n`** as the input argument. The output will be a nested list, with sublists of size **`n`** (except possibly the last sublist which might have lesser than **`n`** elements).

For example, for this case:

```
a=[2,3,4,2,5,1,6,7,8,10,22,3,13]
print(splitList(a,4))
```

The output will be:

```
[[2, 3, 4, 2], [5, 1, 6, 7], [8, 10, 22, 3], [13]]
```

[4] Write a Python program to Merge two given lists of sublists in a way that each sublist on corresponding index merges together. Again, do it by creating a function named as **`mergeSubLists`** having two input arguments (assuming both are lists of equal number of sublists) and output as a bigger list. See the example of input and output here:

Original lists:

```
[[1, 3], [5, 7], [9, 11]]
[[2, 4], [6, 8], [10, 12, 14]]
```

Merged list:

```
[[1, 3, 2, 4], [5, 7, 6, 8], [9, 11, 10, 12, 14]]
```

[5] Below is the detail of marks obtained by 10 students in 5 different subjects:

Subject → / Roll Number ↓	Calculus	Algebra	Programming	Electronics	Statistics
1	89	91	68	88	93
2	78	79	87	78	67
3	94	83	69	79	82
4	67	78	77	82	66
5	88	82	87	77	69
6	93	55	90	82	67
7	76	69	86	75	94
8	66	77	67	64	83
9	82	79	83	71	68
10	59	83	88	84	79

Write a program to display the details of a subject **(Average, Maximum and Minimum)** or a student **(Average, Maximum and Minimum)** entered by user. First store the above values in 2-D list and then complete the remaining logic.

Sample output is:

```
Which detail you want (Subject/Student)
Press 's' for subject and 'r' for student: r

Enter roll number of student: 7
Roll number 7 obtained average marks: 80
Roll number 7 obtained highest marks 94 in subject Statistics.
Roll number 7 obtained lowest marks 69 in subject Algebra.

Want another query (y/n): y
Which detail you want (Subject/Student)
Press 's' for subject and 'r' for student: s

Which subject detail you want?
Press 0,1,2,3 or 4 for Calculus, Algebra, Programming, Electronics or
Statistics
2
```

```
In programming average score is: 80.2
Highest Marks are 90 obtained by roll number 6.
Lowest Marks are 67 obtained by roll number 8.

Want another query (y/n): n
```

The basic structure of the program is given below:

```
classmarks=[
[89,91,68,88,93],
[78,79,87,78,67],
[94,83,69,79,82],
[67,78,77,82,66],
[88,82,87,77,69],
[93,55,90,82,67],
[76,69,86,75,94],
[66,77,67,64,83],
[82,79,83,71,68],
[59,83,88,84,79]]
print("Which detail you want (Subject/Student)")
option=input("Press 's' for subject and 'r' for student: ")
if(option=='s' or option=='S'):
    print("Which subject detail you want?")
    sub=eval(input("Press 0,1,2,3 or 4 for \
Calculus, Algebra, Programming, \
Electronics or Statistics: "))
    #Consider the column in variable sub and find max, min and avg
if(option=='r' or option=='R'):
    roll=eval(input("Enter roll number of student: "))
    #Consider the row in variable (roll-1) and find max, min and avg
```

[6] Determinate of a 3x3 Matrix

We can create a function for determinant of a 2x2 Matrix as shown here:

```
def det2(A):
    'Returns the determinant of 2x2 Matrix A'
    return (A[0][0]*A[1][1])-(A[0][1]*A[1][0])
### Main Program Starts here ###
B=[[2,4],[3,1]]
print(f"Determinant of B is:{det2(B)}")
```

Your task is to create a function named as **`det3()`** that will have one 3x3 matrix as input parameter and will return the determinant of that matrix. You must use **`det2()`** function inside it. Use the function in main program to display the determinant of any 3x3 matrix.

[7] Inverse of a 2x2 Matrix

As you know that the inverse of a matrix is Adjoint of matrix divided by its determinant. You are required to create a function named as **`inv2()`** with one 2x2 matrix as input and it should return

inverse of that. Inside this function use **`det2()`** function for determinant. Moreover, create another function named as **`dispMat()`** with one matrix as input argument and this must display the matrix as indicated at the start of this lab session.

The structure of the program is given as:

```
def det2(A):
    'Returns the determinant of 2x2 Matrix A'
    return (A[0][0]*A[1][1])-(A[0][1]*A[1][0])
def inv2(A):
    'Returns the inverse of A'
    #Complete Logic here
def dispMat(A):
    'Displays matrix A of any size'
    #Complete Logic

### Main Program Starts here ###
B=[[2,4],[3,1]]
print("Inverse of B is: ")
dispMat(inv2(B))
```

[8] Tic-Tac-Toe Game

Write a program for a Tic-Tac-Toe game between two players. You will show a 3x3 grid to the players with empty slots. Player1 will use symbol **(X)** and player2 will use **(O)**. Ask players to select the slot number and fill that with **X** or **O** depending upon the player having the turn. At any stage if the player wins, terminate your program and display the result.

How to do this:

i. Firstly, the game table can be defined as 3x3 list. The initial values in the list can be numbers 1-9 describing the slot numbers. Later these values will be replaced by **O** or **X** depending upon the player and the selected slot. So, the list is defined as:

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

ii. Next, we have to display this grid on the screen. This is something we have to do repeatedly after each turn. Therefore, it is better to define a function that will display the **`grid`** of main program. It is given here:

```
def displayGrid():
    print(f'{grid[0][0]} | {(grid[0][1])} | {(grid[0][2])}')
    print("----------")
    print(f'{grid[1][0]} | {(grid[1][1])} | {(grid[1][2])}')
    print("----------")
    print(f'{grid[2][0]} | {(grid[2][1])} | {(grid[2][2])}')
```

To display current status of the grid at any stage of the game we can simple call above function.

For example, the start of the main program will be:

```
grid=[[1,2,3],[4,5,6],[7,8,9]]
displayGrid()
```

iii. The next step is to take inputs from player1 and player2 repeatedly. This can be done using a `while` loop in main program that will keep iterating until there is a winning condition or the complete table is filled with no result. But to do that an important factor is to set the correct list location with **X** or **O**. User will be entering 1-9 as slot number and it will be your job to convert that number into valid list location.
It will be better to do this by creating a function. The function will have two input arguments; one slot number and the other will be **X** or **O**. The function is given here:

```
def fillSlot(s,mark):
    global grid
    c=(s%3)-1
    r=abs(s-1)//3
    if(grid[r][c] not in [1,2,3,4,5,6,7,8,9]):
        return False
    grid[r][c]=mark
    return True
```

See carefully that the above function takes into account the possibility of wrong slot number and also the possibility of the slot that is already filled by **X or O.** Therefore, this function has one output argument as `True` or `False`. The function will return `False` if the slot number is invalid and will return `True` otherwise.

You will take input from player and will user above function as:

```
print("Player 1 Turn")
p1=eval(input("Enter the slot number:"))
fill=fillSlot(p1,'X')
```

For palyer2 you will pass `'O'` instead of `'X'` and in both cases use `fill` variable to complete the logic.

iv. Next, we must have a function named as `isWinner()` to test the win condition. You have to write this function with no input argument. You will use `grid` values in this function, and this should return `True` or `False`. Note carefully that you have to write this function in a way that it should be independent of player1 or player2. It must simply return `True` if there is wining condition.

v. Complete the `while` loop logic of step iii in a way that you will keep taking turns from player1 and player2. After each turn, call `isWinner()` function and break the `while` loop if it returns `True`. Otherwise, the `while` loop must iterate at least nine times and there is no winning condition in all nine turns you must display game ended as draw.

vi. User clear screen function appropriately so that previous `grid` disappears in each turn.

[9] Game of Life

The Game of Life was developed by John Conway in 1970 to simulate **cellular automaton**. The game board is a 2-dimensional grid. The game has an initial state specified, then proceeds by evolving the grid through **"generations".** Each cell in the grid can be either **"live"** or **"dead"**. Given an initial state of the board, the next generation of the board is determined by the status of each cells' neighbors which include cells directly horizontally, vertically, and diagonally adjacent. An internal cell has 8 total neighbors. The evolution of the board is created by applying the following rules to all cells simultaneously:

i. A live cell with fewer than 2 live neighbors dies.
ii. A live cell with more than 3 live neighbors dies.
iii. A live cell with 2 or 3 live neighbors lives on to the next generation.
iv. A dead cell with exactly 3 live neighbors becomes a live cell.

Here is an example initial 5 x 5 board, and the 1st generation where "*" is live and "." is dead:

```
. . . . .
. * * * .
* * * . .
. . . . .
. . . . .

. . * . .
* . . * .
* . . * .
. * . . .
. . . . .
```

Your program should start with 5x5 grid having 6 live cells and remaining 19 dead, all placed randomly. Next generation should be displayed after a short time. Your program should display next 20 generations.

For further detail, visit:

http://en.wikipedia.org/wiki/Conway's_Game_of_Life