⇒ Practicing is the key. → the better you become.

⇒ "Able to solve the problem but not getting confidence of solving a new question if faced."

Introduction to Multi-dimensional Arrays

2D Array

	Column 0	Column 1	Column 2
Row 0	X[0][0]	X[0][1]	X[0][2]
Row 1	X[1][0]	X[1][1]	X[1][2]
Row 2	X[2][0]	X[2][1]	X[2][2]

2D.

Array

Multi-Dimensional

3D Array

high level.

	Column 0	Column 1	Column 2
Row 0	000	001	002
Row 1	010	100	101
Row 2	020	110	102
	120	200	201
		210	211
		220	221

Array 0 | Array 1 | Array 2 | 202 | 212 | 222

1-D →

same type of data stored.

2-D Array :

Problem Statement : I want to store distances between some cities

How ? ← I will tell.

Why ??

Kanpur → K → B → 100 km

Bokaro → B → L → 200 km

Lucknow → L → C → 50 km

Chennai → C → L → 150 km

Jodhpur → J → K → 300 km

Appr. 1 → Dictionary.

Kanpur → K → B → 100 km

Bokaro → B → L → 200 km

Lucknow → L → C → 50 km

Chennai → C → B → 150 km

Jodhpur → J → K → 300 km

{ K-B : 100,
B-L : 200,
L-C : 50,
C-B : 150
J-K : 300
}

→ Advantages
easy to fetch., no duplicates.

Appr 2 : 2D Array.

Kanpur → K → B → 100 km

Bokaro → B → L → 200 km

Lucknow → L → C → 50 km

Chennai → C → B → 150 km

Jodhpur → J → K → 300 km

	K	B	L	C	J
K	0	100	0	0	300
B	100	0	200	150	0
L	0	200	0	50	0
C	0	150	50	0	0
J	300	0	0	0	0

Approach 1 : using Dictionary

```
# Creating a dictionary to store distances between cities
distances = {
    "CityA_CityB": 100,
    "CityA_CityC": 150,
    "CityA_CityD": 200,
    "CityB_CityC": 50,
    "CityB_CityD": 250,
    "CityC_CityD": 100,
}
```

```
# Accessing the distance between City A and City D
print("Distance between City A and City D:", distances["CityA_CityD"],
"km")
```

Pros:

- Direct mapping between two cities and their distance.
- Easy to access the distance between any two given cities.

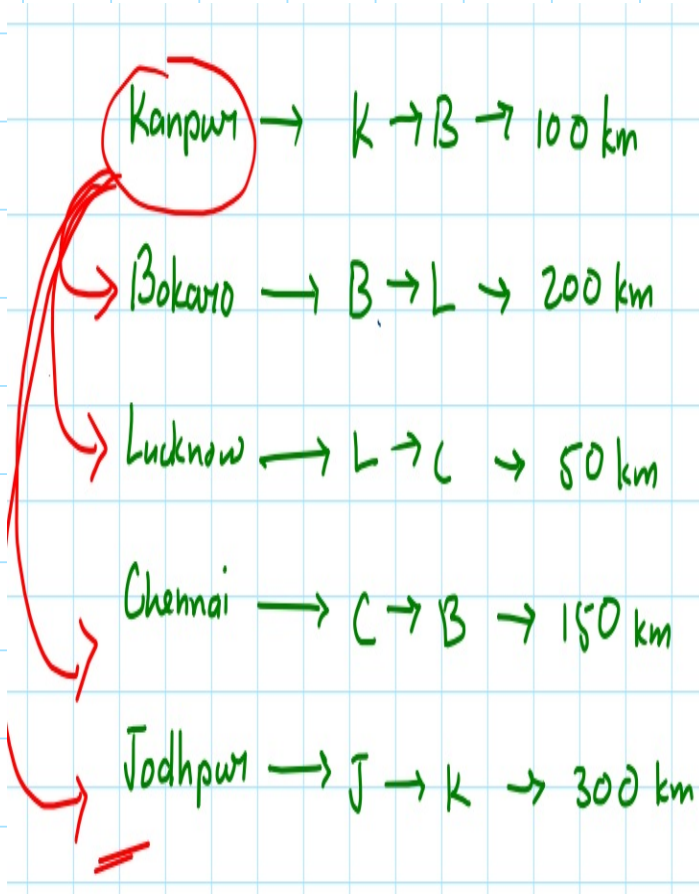
Cons:

- Can become cumbersome as the number of cities grows, leading to a significant increase in dictionary keys.
- Not the most space-efficient if many distances are the same or zero.

Approach 2 : 2D Array

- A more structured way to represent such data is by using 2D arrays (lists of lists in Python), where each row and column represent a city, and the intersection represents the distance between those cities.

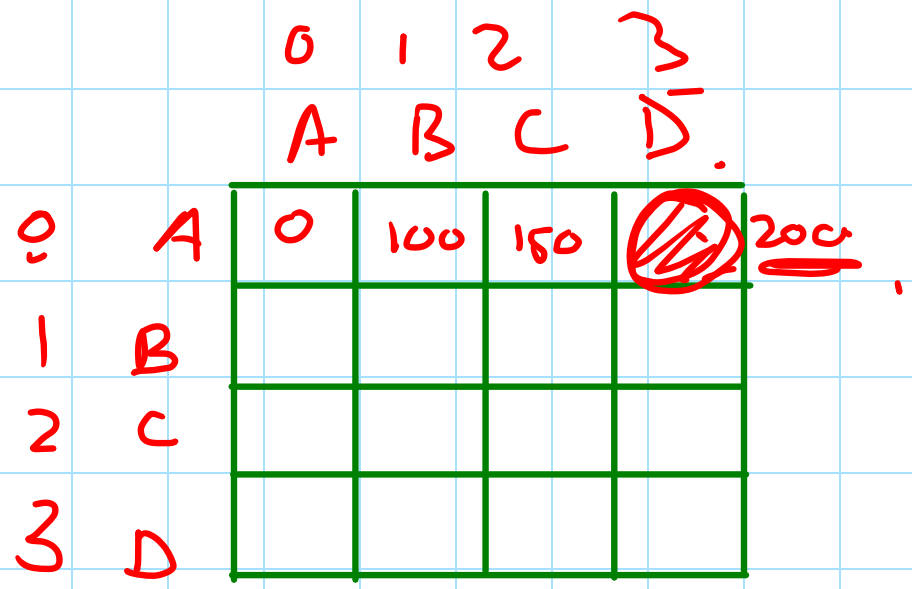
	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]



	K	B	L	C	J
K	0	100	0	0	300
B	100	0	200	150	0
L	0	200	0	50	0
C	0	150	50	0	0
J	300	0	0	0	0

Approach 2 : 2D Array

A more structured way to represent such data is by using 2D arrays (lists of lists in Python), where each row and column represent a city, and the intersection represents the distance between those cities.




```
# Creating a 2D array to store distances between cities
# Assume CityA=0, CityB=1, CityC=2, CityD=3 for indexing purposes
distances = [
    [0, 100, 150, 200],  # Distances from City A
    [100, 0, 50, 250],   # Distances from City B
    [150, 50, 0, 100],   # Distances from City C
    [200, 250, 100, 0]   # Distances from City D
]

# Accessing the distance between City A and City D
print("Distance between City A and City D:", distances[0][3], "km")
```

Approach 2 : 2D Array

A more structured way to represent such data is by using 2D arrays (lists of lists in Python), where each row and column represent a city, and the intersection represents the distance between those cities.

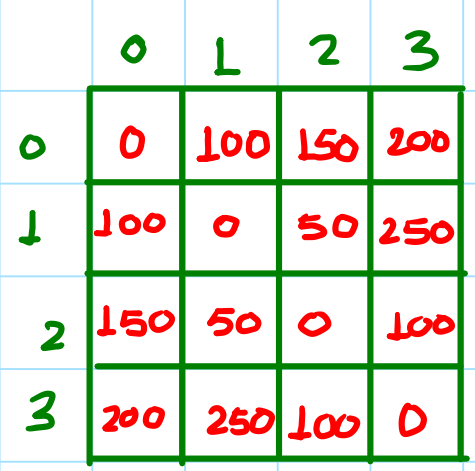
```
# Creating a 2D array to store distances between cities
# Assume CityA=0, CityB=1, CityC=2, CityD=3 for indexing purposes
distances = [
    [0, 100, 150, 200],  # Distances from City A
    [100, 0, 50, 250],   # Distances from City B
    [150, 50, 0, 100],   # Distances from City C
    [200, 250, 100, 0]   # Distances from City D
]

# Accessing the distance between City A and City D
print("Distance between City A and City D:", distances[0][3], "km")
```

Pros:

- Offers a structured, tabular form of data representation.
- Efficiently stores distances, especially when the number of cities is fixed and known.

Cons:

- Can require a significant amount of space, particularly if there are many cities with zero or the same distances.



Dictionary Vs 2D Array :

Both dictionaries and 2D arrays offer viable methods for storing distances between cities.

The choice between them depends on the specific requirements of your problem and your preference for space efficiency versus ease of data access.

Beginners might find dictionaries more straightforward for smaller sets of data, while 2D arrays offer a more scalable solution for larger datasets.

Visualizing Report Cards :

Math → M

Biology → B

Economics → E

100 students in a class learning above subjects.

→ what would be the efficient way of storing / representing their scores.

Dictionary :-

{ Ram : {M : 50, B : 40, E : 60}, - - - - - .

:

:

:

:

3

not efficient in terms of space.

(keys)

Visualizing Report Cards :

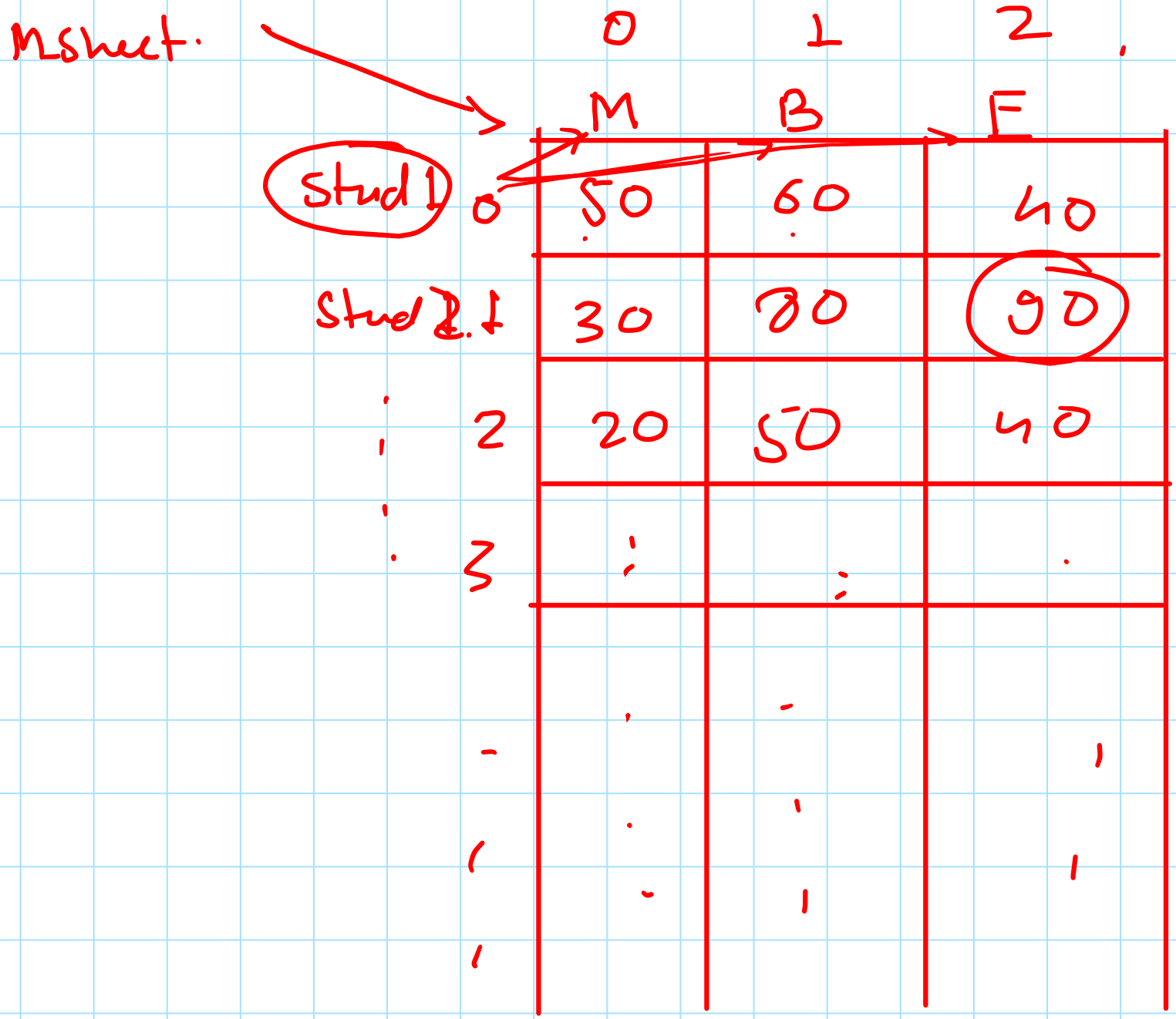
Math → M

Biology → B

Economics → E

100 students in a class learning above subjects.

→ what would be the efficient way of storing / representing their scores.



mSheet[1][2] → 90.

```
# Example 2D array
grades = [
    [90, 80, 70],  # Grades for student 1
    [85, 75, 65],  # Grades for student 2
    [88, 78, 68]   # Grades for student 3
]
```

Subjects

Students

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Indexing in 2D Arrays :

Arr

Symmetric.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Rows → 3

Columns → 3.

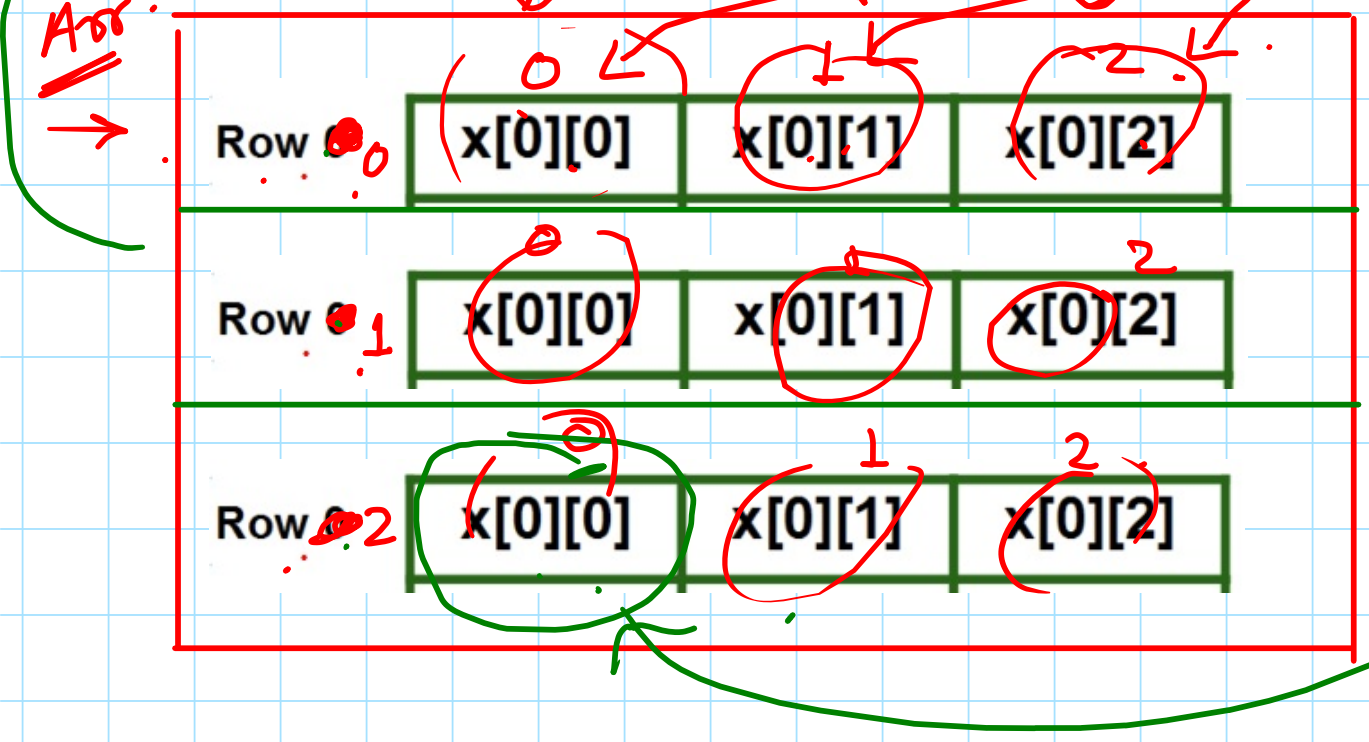
len(Arr)

↳ # of rows.

# of columns.

↳ len(Arr[0])

2



Row[0] Row[1] Row[2]

Arr[2][0]

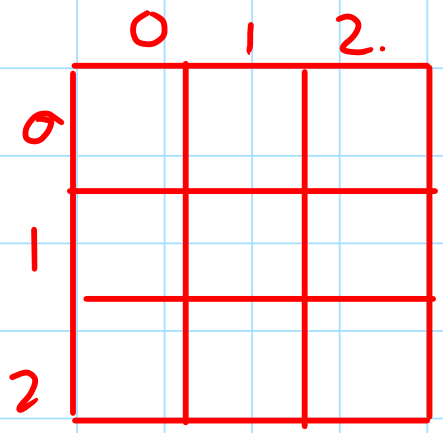
Arr.length() → # of rows.

Arr[0].length() → # of columns

Finding the Length of Rows and Columns in 2D Arrays :

3 x 3

grades.



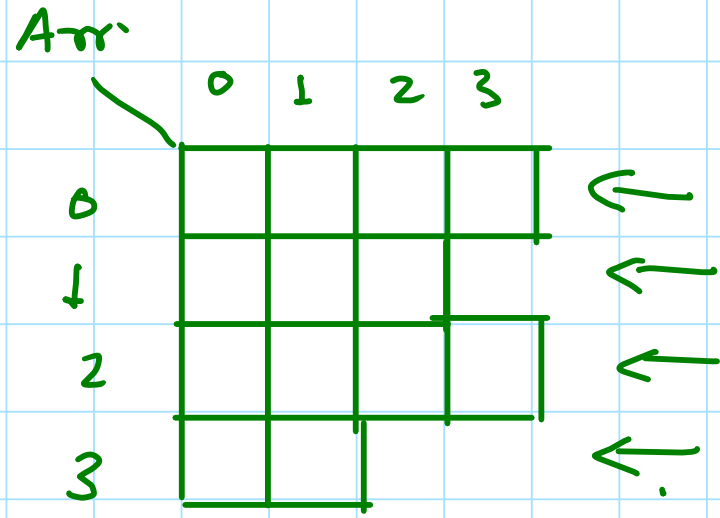

```
# Example 2D array
grades = [
    [90, 80, 70],  # Grades for student 1
    [85, 75, 65],  # Grades for student 2
    [88, 78, 68]   # Grades for student 3
]

# Finding the number of rows
num_rows = len(grades)
print("Number of Rows:", num_rows)

# Finding the number of columns
num_columns = len(grades[0])
print("Number of Columns:", num_columns)
```

→ # of rows or # of 1-D array.

3

→ # of col or length of 1-Darray.

3



len(Arr[0]) → 4

len(Arr[1]) → 3

len(Arr[2]) → 4

len(Arr[3]) → 2

Traversing 2D Arrays :

x.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

x[0][0] → x[0][1] → x[0][2]

row col.

```
for (i=0 ; i<len(x) ; i++)
    for (j=0 ; j<len(x[0]) ; j++)
        {
        print ( x[i][j] )
        }
```

rows.

i<3 ✓ 1<3 ✓ 2<3 ✓ 3<3 ✗

3 < 3x 0<3 ✓

outside → rows are repeating.

inside → columns are repeating.

x →

	0	1	2
0	5	10	50
1	4	20	70
2	100	90	80

o/p.

5 10 50 4

20 70 100

90 80

i = 0 , j = 0, 1, 2, 3x = x[0][0]

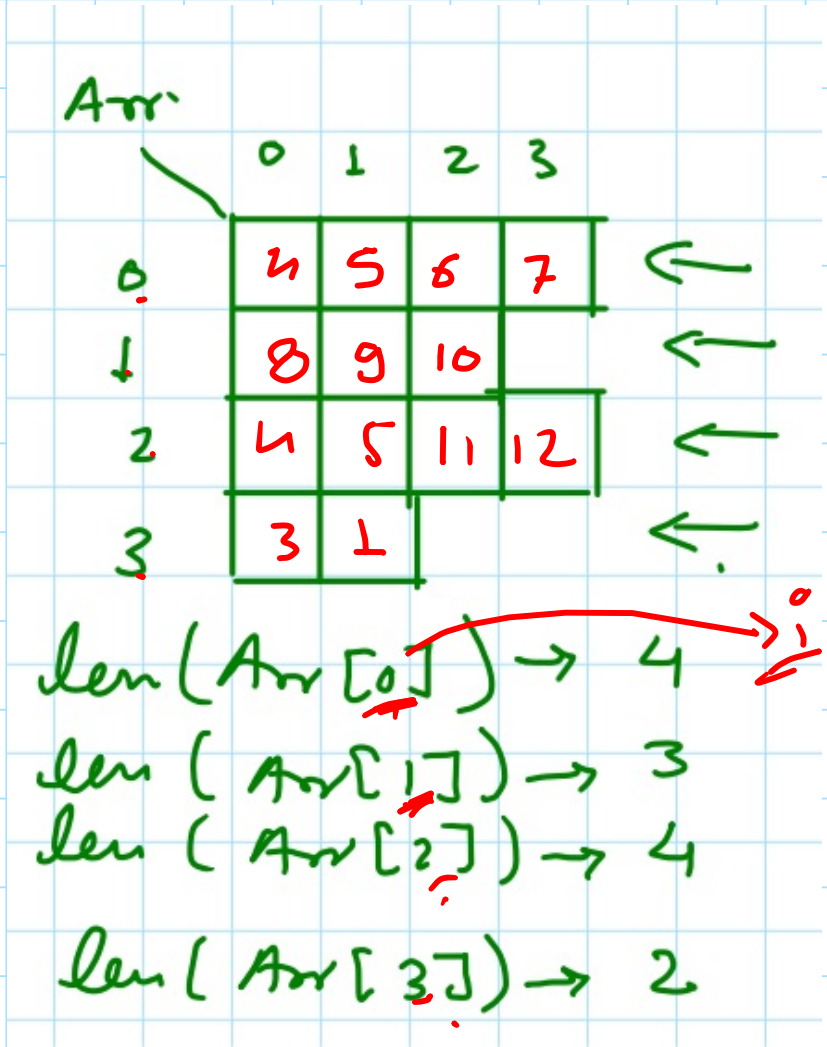
i = 1 , j = 0, 1, 2, 3x

i = 2 , j = 0, 1, 2, 3x

i = 3x

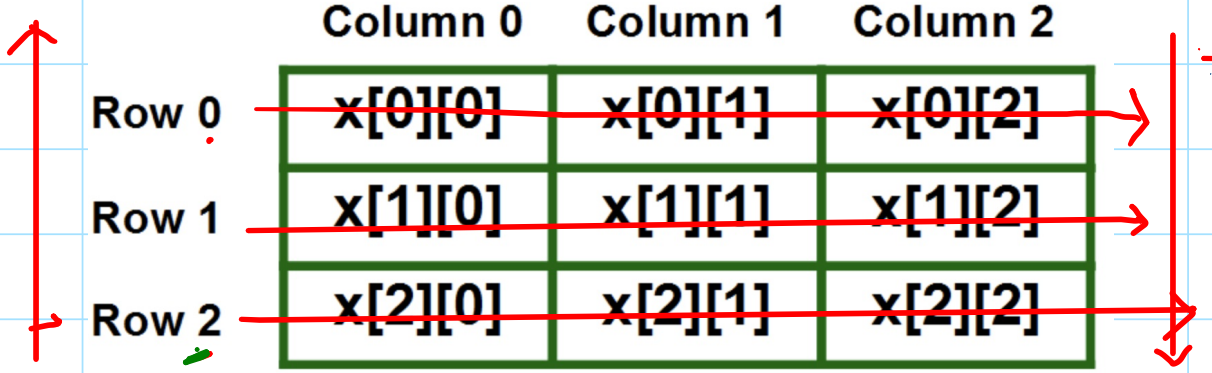
print all the items in array

```
for( i=0 ; i<len(Arr) ; i++)
{  for( j=0; j<len(Arr[i]) ; j++)
   {
   }
}
```



5 minutes.

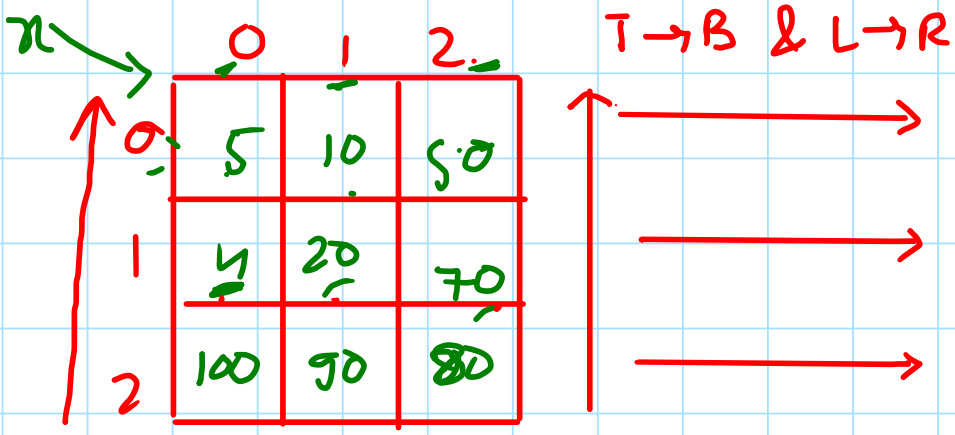
back at 1:05.



T → B & L → R. →

B to T & L to R.

100	90	80
4	20	70
5	10	50



row → outer loop → reversed.

col → inner loop → forward.

Nested Loop.

```
for( i = len(x)-1 ; i >=0 ; i--)//reverse loop
{
  for(j = 0 ; j < len(x[0]) ; j++)
    {   print( x[i][j] )
    }
}
```

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

x [row ind] [col ind]

Column wise Traversal : (Reverse order)

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

arr:

	0	1	2
0	5	10	50
1	4	20	70
2	100	90	80

o/p → 100 4 5
90 20 10
80 70 50

len(arr) - 1 → # of rows.

Very important:

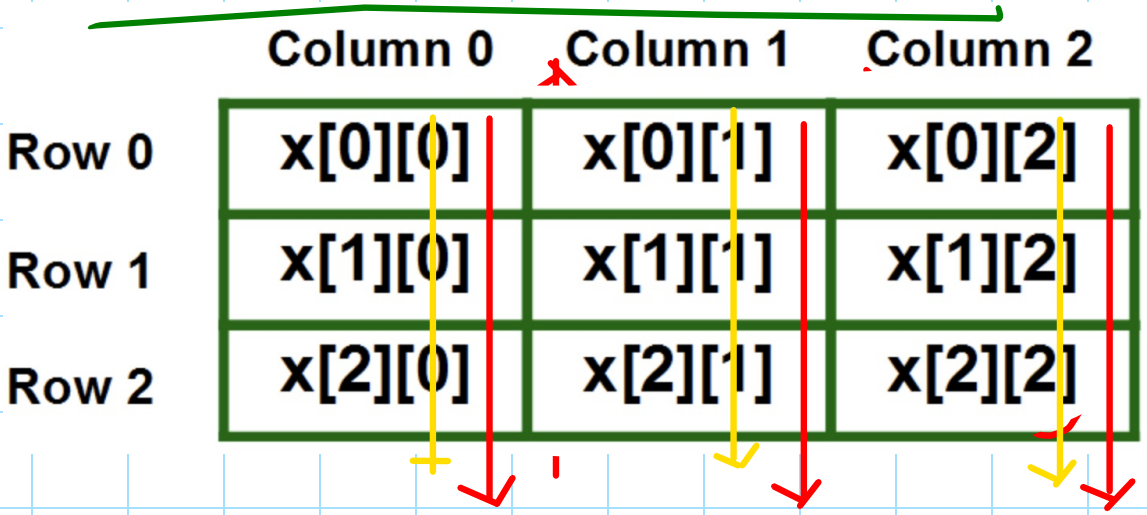
- → outer loop → colums → forward.
- → inner loop → rows → reverse.

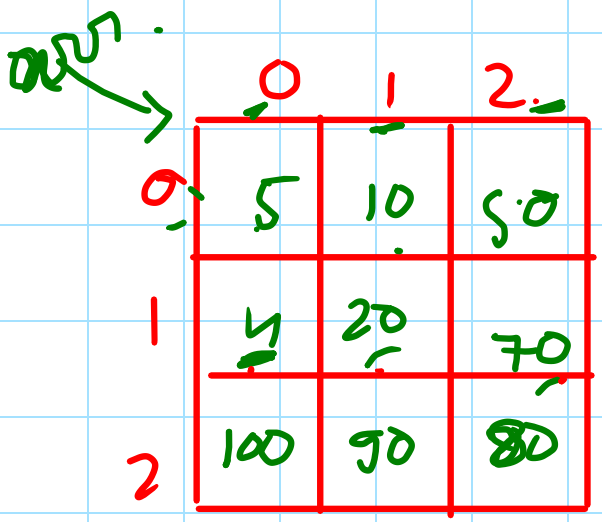
```
for(i=0 ; i< len(arr[0]); i++)
{
    for(j = len(arr)-1 ; j>=0; j--)
    {
                  row col
        print( arr [j][i] )
    }
}
```

HW: i=0 , j = arr[0,0]
arr[0,0]

Column wise Traversal : [forward order.

o/p → 5 4 100
10 20 90
50 70 80



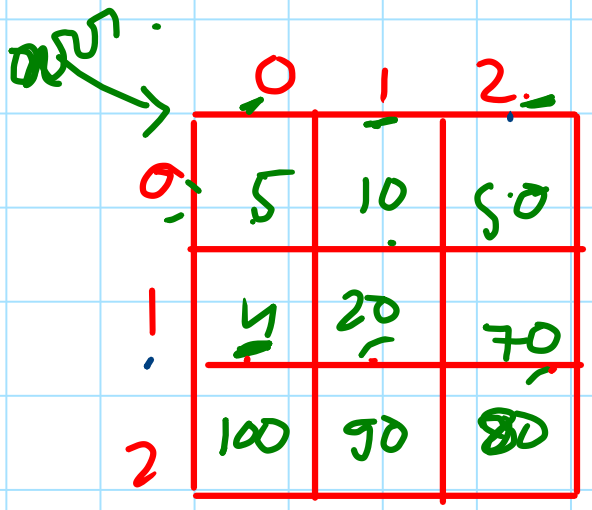


Code (M/W)

outer loop → column → forward.

inner loop → row → forward

Finding a Target item in 2D array :-




```
key = 70

for (i=0; i< len(arr); i++)
  { for (j=0; j< leng(arr[0]); j++)
    { if (arr[i][j] == key)
        return true
    }
  }
return false.
```

Printing Diagonal items:

sq. matrix

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

L to R.

Right to left ~~X~~

o/p → 5 20 80

	0	1	2
0	5	10	50
1	4	20	70
2	100	90	80

L to R.

5 → [0][0]

20 → [1][1]

80 → [2][2]

outer →

inner →

```
#of loops = 2
for(i=0 ; i<3; i++)
{for (j=0 ; j <3; j++)
{  if (i==j)
     print arr[i][j];
}
}
```

i = 0, 1, 2, 3

j = 0, 1, 2, 3

```
for( i=0, j=0; i<len(arr); i++, j++)
{  print (arr[i][j])
}
```

a[0][0] ✓

a[1][1] ✓

a[2][2] ✓

H.W

print right Diagonal.

sq. matrix.

L to R.

R

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

arr

	0	1	2
0	5	10	50
1	4	20	70
2	100	90	80

arr[0][2]

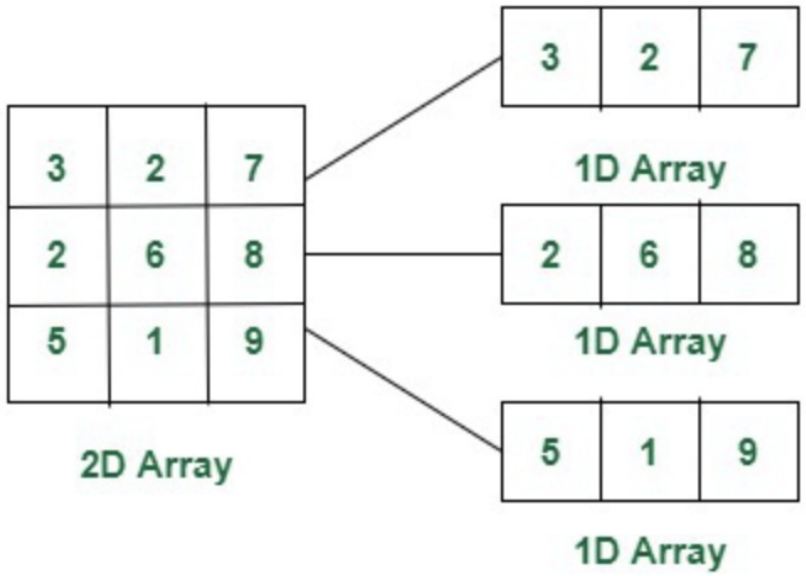
arr[1][1]

arr[2][0]

row → 0 to 2.

col → 2 to 1.

Can you do it using single loop?



2D Array is a collection of 1D Arrays