2D lists.

l = [1, 2, "abc", True]

↓

normal list

l = [[1, 2] , [2, 3]]

↓

2D array in python

↓

list of lists in python

l = [1, 2, [3, 4]]

```
>> l[0] = 1
>> l[1] = 2
>> l[2][0] = 3
>> l[2][1] = 4
```

*imp. to be careful while accessing elements in 2D lists, bec. of diff. in elements in constituent lists

↓

similarly we can even create 3D arrays, 4D arrays (lists) in python (list → list → list)

alternate methods of creating 2D lists.

l2D = [[i, i+1, i+2] for i in range(10)]

↓

(2D array)

l2D1 = [[~~for~~ j for j in range (5)] for i in range (10)]

↓

10 rows, 5 column list with numbers from 0 – 4.

```
l2D2 = [[5*i + j for j in range(5)] for i in range(10)]
```

↓

multiple of 5's as beginning indices.

↓

(0 → 49)

How to take input in 2D array.

* first we need to know how many rows and columns to read in input (take that also as i/p)

↓

to get r,c in single line

```
str = input().strip().split(" ")
r = int(str[0])
c = int(str[1])
```

↓

* Now we need to accept entries for the 2D array

↓

we accept a list of numbers and then reshape into a multi-dimensional array

↓

(get all numbers, and convert to 2D array)

for actual array

```
l = [int(i) for i in input().strip().split(" ")]
```

↓

```
l2 = [[l[j] * c + i] for i in range(c)] for j in range(r)]
```

↓

Method 1 (for single line input)

or

```
l2 = []
for i in range(r):
    l2.append([])
```

→ create a new list at each row number

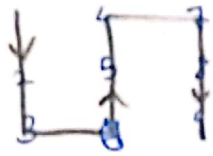
```
for j in range(m):
    out[i].append(l[i*m+j])
```

↓

for each list of each row, we insert consequent element from l

* wave print

if index of columns even, (read top-bottom)
(if odd, read bottom-top)



↓

printing them in 1 line

↓

1 2 3 6 5 4 7 8 9

```
r = len(out)
c = len(out[0])
```

} to find no. of rows and columns in 2D array

```
for j in range(c):
    if j%2==0:
        for i in range(r):
            print(out[i][j], end=" ")
    else:
        for i in range(-1, -r-1, -1):
            print(out[i][j], end=" ")
```

← we can use while loop here to avoid confusion

* we use print() to print or move to next line

or even: for i in range(r-1, -1, -1)
from l[r-1] to r[-1]

(goes till 0)

↓ won't be included

* largest row column

↓

store row sums and columns in another 2D array in 0th and 1st row resp.

↓

use maxv variable to check max value and then accordingly print result

↓

normal input taking mechanism

↓

```
out 1 = [[], []]

for i in range(r):
    out[0].append(0)
    for j in range(c):
        out[0][i] += out[i][j]

for j in range(c):
    out[1].append(0)
    for i in range(r):
        out[1][j] += out[i][j]

maxv = 0
maxn = 0
maxc = 'R'

for i in range(r):
    if maxv < out1[0][i]:
        maxv = out1[0][i]
        maxn = i
        maxc = 'R'
```

→ initially the first out list is empty, to put in sums, we need the 0 there

(26) #importing modules in python
↓
(analogous to libraries in other languages)

import x
import x as ____ → alias

** to import a part of the module, we use:

from x import y
↓
import math as m
~~print(sqrt(10))~~
print (m.sqrt (10))

→ numpy → library for python, adding support for large, MD arrays, and matrices, along with large collection of high level maths functions on these arrays.

* python lists are slow : (a) can store anything, hence not stored in a compact way,

↙ reasons for using numpy

for ex. in c++, if we have an array of 20 integers, we store them in contagious memory slots.
↓
compact manner
↓
better management of memory
↓
more efficiency
↓
in python, we can store or replace anything for anything, hence compact storing is not an option.

b) python itself is slower than C/C++, hence lists automatically get slower

↓

Numpy → code written in C ↳ automatically fast

↓

module used for arrays

↓

same type of elements (as in C)

↓

(stored compactly)

import numpy as np. → (importing numpy modules)

l = [1, 2, 3]

np.array(l) → numpy array created

↓

if one of the elements of used list is a string, then the numpy array is stored as strings completely.

np.zeros(10) → array of 0s

np.ones(10) → array of 1s

⇒ a. = ~~np.zeros~~ np.zeros(10)

a.dtype → datatype of elements of this array

↳ 'float64'

↳ elements formed using zeros, ones.

↓

a = np.zeros(10, int)

↓

we can specify the underlying type we would prefer the elements to occupy.

creating 2D numpy array

np.zeros ((2, 3)) → passing dimension as tuple

multi-dimensional array

b = np.zeros ((2, 3, 2))

>> a = np.zeros (10, int).

>> a[0] = '1' → converts this to 1 (integer)

↓

if given a or other string, then error, as it won't be convertible.

>> a.dtype

* normal list accessing methods.
* slicing

a [2 : 4]

* shape [analogous to size for MD arrays]

b. shape → (2, 3, 2) → tuple

↓

can be used for loops, etc.

* creating array for a range

~~np.arange~~ np.arange (10).

↓

array ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

* a = np.array ([0, 1, 2, 3])

a.shape → (4,)

```
a = np.array([[1,2,3], [4,5,6]])
a[0]
a[:, 0]
```

↓ everything in terms of rows, 0th column

↓

```
a[0:2, 0]
```

↳ 0 to 2nd row, and 0th column

↓

(any subarray can be accessed)

numpy operations

numpy arrays behave like mathematical vectors.

```
a = np.array([1, 2, 3])
b = np.array([5, 4, 6])
```

a + 2 → element wise operations

↓

scalar addition to vector

a + b → if dimensions are different

↓

(error)

vector addition

```
a * 2
a * b.
a ** 2
a ** b.
```

element wise operations, mapping

```
a.mean()
a.sum()
```

* Numpy arrays make lot of calculation easy for us

$\overset{n}{a}$ $\overset{n}{b}$

$\begin{bmatrix} \\ \\ \end{bmatrix}$ $\begin{bmatrix} \\ \\ \end{bmatrix}$

n dimension space

↓

a and b are 2 different points in that space

↓

sq. distance b/w them

↓

L2 norm

↑ p1 ↑ p2

$(a-b)^{**2}.sum()$

$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$ $\begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$

$$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

↓

square distance (sd) = $((a-b)^{**2}).sum()$

d = m.sqrt(sd)

* $\Sigma(x_i - x_{mean})^2$ → how far are a set of points from the mean

$((a - a.mean())^{**2}).sum()$

* a = np.array([[1,2],[3,4]])
  b = np.array([[5,6],[7,8]])

  a.dot(b)

  → allows dot product of arrays.

* np.matrix()

  → numpy matrices. → behaviour little different from numpy arrays

  ↓

  subclass of numpy arrays

  ↓

  strictly 2D.

  ↓

  (behave more like matrices)

  ↓

  i.e. $\underbrace{a * b}_{\text{(product)}}$

* a = np.array([0, 1, 2, 3])
  b = np.array([2, 3, 4, 5])

  a.dot(b) ⇒ 26

* a = np.matrix('1 2 4 ; 3 4 5')
  a.shape ⇒ (2, 3)

* np.sum([[0,1],[0,5]], axis=1)
  array([1, 5])

  0 1 → 1
  0 5 → 5

  axis=0
  0
  6