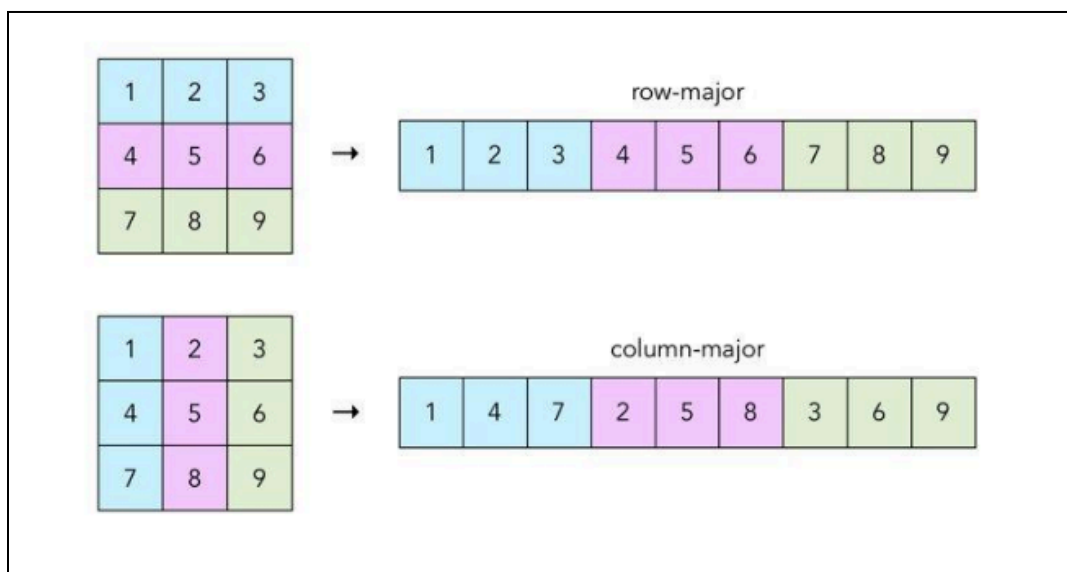


Multidimensional Array

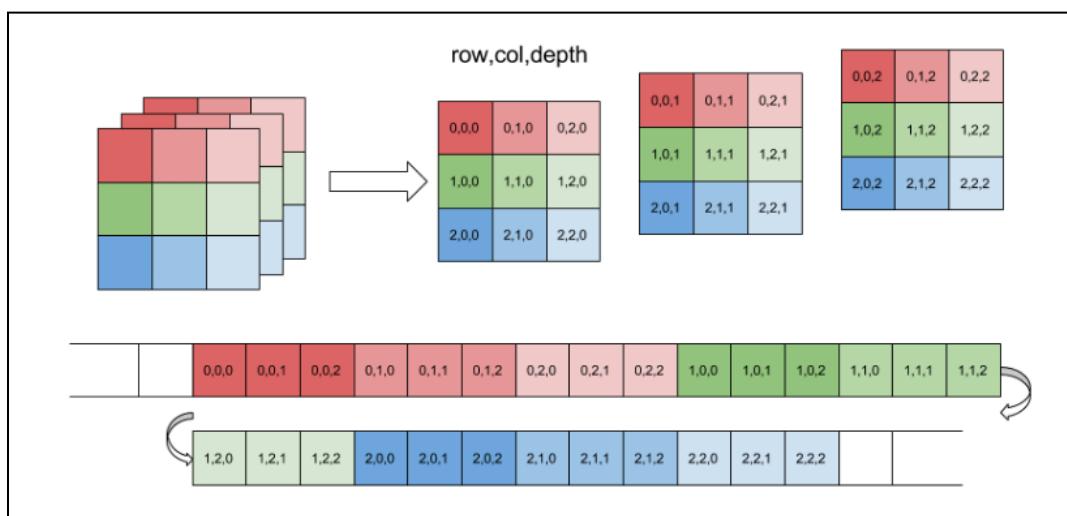
Multidimensional arrays give us a natural way to deal with real-world objects that are multidimensional, and such objects are quite frequent.

Programming languages store multidimensional arrays as linear arrays in the RAM as RAM is a linear array of memory cells.

2D to 1D Array:



3D to 1D Array:



We will stick to row-major ordering, as that is more common in application programming.

Conversion of Linear and Multidimensional Index:

If we have a 4D array with dimensions $M \times N \times O \times P$, and we want to access an element at $\text{arr}[w][x][y][z]$. Then the index of that element in the linear array is:

$$\text{Linear Index} = w \times (N \times O \times P) + x (O \times P) + y \times P + z$$

Let A be a 4D array with dimensions $5 \times 2 \times 4 \times 3$

What is the linear index of $A[2][1][3][2]$?

$$\begin{aligned} \text{Ans: Linear Index} &= 2 \times (2 \times 4 \times 3) + 1 \times (4 \times 3) + 3 \times (3) + 2 \\ &= 71 \end{aligned}$$

Follow-Up Question:

Q) How to calculate a Multidimensional Index from a Linear Index? Let us consider an example but in reverse.

Let A be a 4D array with dimensions $5 \times 2 \times 4 \times 3$,

What is the multidimensional index of 71?

Ans:

$$71 = w \times (2 \times 4 \times 3) + x \times (4 \times 3) + y \times (3) + z$$

$$\begin{aligned} w &= 71 // (2 \times 4 \times 3) = 2, \\ 71 \% (2 \times 4 \times 3) &= 23 \end{aligned}$$

$$\begin{aligned} x &= 23 // (4 \times 3) = 1 \\ 23 \% (4 \times 3) &= 11 \end{aligned}$$

$$\begin{aligned} y &= 11 // (3) = 3 \\ 11 \% (3) &= 2 \end{aligned}$$

$$z = 2$$

Therefore, Multidimensional Index = $[2][1][3][2]$

Multidimensional Array Declaration/Initialization:

- Array name
- Type of data to be stored in Array
- Size of Array

```
import numpy as np

# Creating a 2D array with zeros of size 4x5
arr = np.zeros((4,5), dtype=int)

# Creating a 2D array with integers
arr = np.array([[1, 2, 3], [4, 5, 6]], dtype=int)
```

Multidimensional Array Operations:

- Read/Write:

```
# Reading the third row and fourth column of a 2D array
arr_2D[2][3]

# Writing the third row and fourth column of a 2D array
arr_2D[2][3] = 'hello'
```

- Iteration:

```
#Printing row-wise
def print_row(arr):
    row, col = arr.shape
    for i in range(row):
        for j in range(col):
            print(arr[i][j])
        print()
```

```
#Printing column-wise
def print_col(arr):
    row, col = arr.shape
    for i in range(col):
        for j in range(row):
            print(arr[j][i])
        print()
```

- Summation:

```
#Summing all elements
def array_sum(arr):
    sum = 0
    row, col = arr.shape
    for i in range(row):
        for j in range(col):
            sum += arr[i][j]
    return sum
```

```
#Summing every row
def row_wise_sum(arr):
    row, col = arr.shape
    result = np.zeros((row, 1), dtype=int)
    for i in range(row):
        for j in range(col):
            result[i][0] += arr[i][j]
    return result
```

4	3	8	→	15
2	5	1	→	8
7	-1	9	→	9
5	4	-2	→	7

#Summing every column

```
def col_wise_sum(arr):
    row, col = arr.shape
    result = np.zeros((1, col), dtype=int)
    for i in range(col):
        for j in range(row):
            result[0][i] += arr[j][i]
    return result
```

4	3	8
2	5	1
7	-1	9
5	4	-2

18	11	16
----	----	----

- Swapping:

#Swapping the two columns of a $m \times 2$ matrix

```
def swap_two_col(arr):
    row, col = arr.shape
    for i in range(row):
        arr[i][0], arr[i][1] = arr[i][1], arr[i][0]
    return arr
```

4 _(0,0)	3 _(0,1)
2 _(1,0)	5 _(1,1)
7 _(2,0)	6 _(2,1)

3 _(0,0)	4 _(0,1)
5 _(1,0)	2 _(1,1)
6 _(2,0)	7 _(2,1)

#Swapping the columns of a $m \times n$ matrix

```
def swap_col(arr):
    row, col = arr.shape
    for i in range(row):
        for j in range(col//2):
            arr[i][j], arr[i][col-1-j] = arr[i][col-1-j], arr[i][j]
    return arr
```

4 _(0,0)	3 _(0,1)	8 _(0,2)	-7 _(0,3)
2 _(1,0)	5 _(1,1)	-1 _(1,2)	12 _(1,3)
-6 _(2,0)	16 _(2,1)	9 _(2,2)	10 _(2,3)
4 _(3,0)	13 _(3,1)	11 _(3,2)	18 _(3,3)

-7 _(0,0)	8 _(0,1)	3 _(0,2)	4 _(0,3)
12 _(1,0)	-1 _(1,1)	5 _(1,2)	2 _(1,3)
10 _(2,0)	9 _(2,1)	16 _(2,2)	-6 _(2,3)
18 _(3,0)	11 _(3,1)	13 _(3,2)	4 _(3,3)

- Addition:

#Adding two matrices of the same dimension

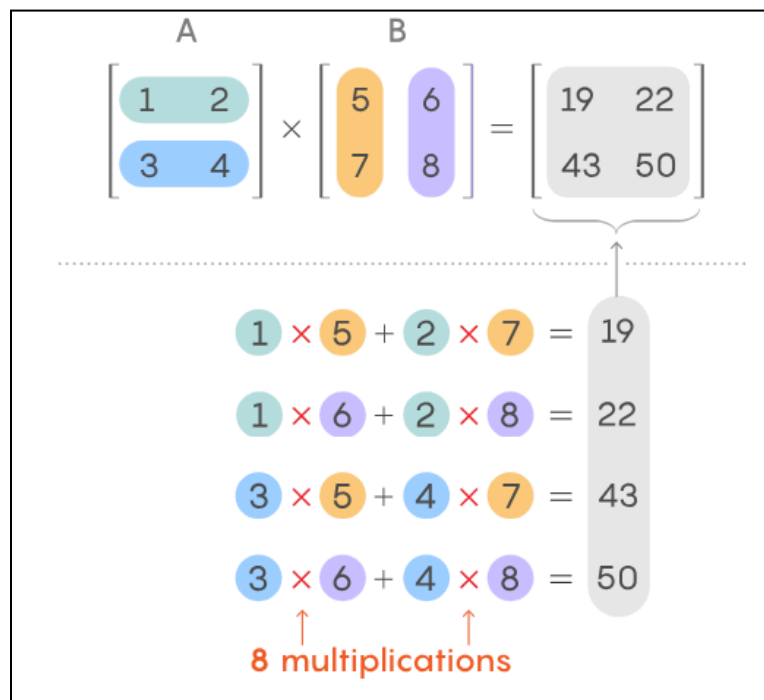
```
def add_matrices(arr1, arr2):
    row1, col1 = arr1.shape
    row2, col2 = arr2.shape
    if row1 != row2 or col1 != col2:
        return "Dimension mismatch"
    result = np.zeros((row1, col1), dtype=int)
    for i in range(row1):
        for j in range(col1):
            result[i][j] = arr1[i][j] + arr2[i][j]
    return result
```

#Adding elements of the primary diagonal in a square matrix

```
def sum_primary_diagonal(arr):
    sum = 0
    row, col = arr.shape
    if row != col:
        return "Not a square matrix"
    for i in range(row):
        sum += arr[i][i]
    return sum
```

4 _(0,0)	3 _(0,1)	8 _(0,2)
2 _(1,0)	5 _(1,1)	1 _(1,2)
7 _(2,0)	6 _(2,1)	9 _(2,2)

- Multiplication



```

#Multiplying a m × n matrix and a n × p matrix
def multiply_matrices(arr1, arr2):
    row1, col1 = arr1.shape
    row2, col2 = arr2.shape
    if col1 != row2:
        return "Cannot Multiply"
    result = np.zeros((row1, col2), dtype=int)
    for i in range(row1): #For each row in arr1
        for j in range(col2): #For each column in arr2
            for k in range(col1): #For multiplying and adding
                result[i][j] += arr1[i][k] * arr2[k][j]
    return result

```