Project Work

Matrices and their Operations in Python

Contents

Abstract

Matrices are a part of Linear Algebra which is used everywhere in Computer Science. It is used in computer graphics to create 2D/3D models, animations, etc. It is used in cryptography (making data secure) by using matrices to store data and a key matrix to encrypt it and its inverse to decrypt it. In this project we will look at operations on matrices in Python.

1 What is a Matrix?

A matrix is a rectangular array of data[1] arranged in rows and columns.
A matrix looks like the following:

$$A = \begin{bmatrix} 1 & 10 & 12 \\ 2 & 20 & 22 \end{bmatrix}_{2\times 3}$$

A matrix is represented by capital letters and the subscript represents Number of rows $\times$ Number of columns and the matrix A is called *a 2 by 3 matrix*.
In Python, to enter a matrix, we use nested lists like:

```
Matrix=[
  [1,10,12],
  [2,10,22]
]
```

1.1 Accessing a Matrix

A matrix a is generally written as $A = [a_{ij}]_{m\times n}$ where $1 \leq i \leq m \wedge 1 \leq j \leq n$. Thus, if we know the location of an element say, *element in row 2 and column 1*, we can write it as a_{21}. In Python as well, if we need to find the *element in row i and column j*, we can return it as:

```
def find_element(matrix, row, column):
  row_index=row-1 #we need to use row-1 as indexes begin from 0
  column_index=column-1
  return matrix[row_index][column_index]
```

Problem. We can also access the elements of a matrix, either row or column wise.

Code. To print the matrix row-wise:

```
def row_wise(matrix):
  for i in range(len(matrix)):
    for j in range(len(matrix[0])):
      print(matrix[i][j],end="\t")
  print("\n")
```

If matrix is $A = \begin{bmatrix} 1 & 10 & 12 \\ 2 & 20 & 22 \end{bmatrix}$, then, the output is,

```
1       10      12

2       20      22
```

To print the matrix column-wise:

```
def column_wise(matrix):
  for i in range(len(matrix[0])):
    for j in range(len(matrix)):
      print(matrix[j][i],end="\t")
  print("\n")
```

The output in this case is:

```
1       2

10      20

12      22
```

1.2 Null Matrix

A null matrix is one such that,

$$a_{ij} = 0 \quad \forall\, i, j$$

Such a matrix is represented by O.

[1] Data may be of any form, like numbers, expressions or alphabets.

Problem. Creating a null matrix of a given order.

Code.

```
def null(rows,columns):
  null_matrix=[[0 for i in range(columns)] for i in range(rows)] #loop over columns then over rows
  return null_matrix
```

1.3 Upper and Lower Triangular Matrices

The upper triangular matrix is a matrix in which all the entries below the diagonal are zero, i.e.,

$$a_{ij} = 0 \quad \forall\, i \geq j$$

Or,

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

The lower triangular matrix is a matrix in which all the entries above the diagonal are zero[2], i.e.,

$$a_{ij} = 0 \quad \forall\, i \leq j$$

Or,

$$A = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

Problem. Creating an upper and lower triangular matrix.

Code. First for an upper triangular matrix,

```
def upper_triangular(matrix):
    rows=len(matrix)
    columns=len(matrix[0])
    upper_matrix=null(rows,columns) #null matrix
    for i in range(rows):
        for j in range(columns):
            if i>=j: #Condition for a null matrix
                upper_matrix[i][j]+=matrix[i][j]
            else:
                continue
    return upper_matrix
```

If the matrix is $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, then, the output is,

```
[[1, 0, 0], [4, 5, 0]]
```

Now, for a lower triangular matrix,

```
def lower_triangular(matrix):
  rows=len(matrix)
  columns=len(matrix[0])
  lower_matrix=null(rows,columns) #null matrix
  for i in range(rows):
    for j in range(columns):
      if j>=i: #Condition for a null
        lower_matrix[i][j]+=matrix[i][j]
      else:
        continue
  return lower_matrix
```

[2]These definitions are open for discussion. Some authors claim that the matrix must be square, while some do not restrict the matrix. Even though a 'triangle' would not be formed in the case of rectangular matrices, it is acceptable. We have chosen not to restrict the matrices to only square matrices.

The output in this case is,

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

1.4 Transpose of a Matrix

The transpose of a matrix $A = [a_{ij}]_{m\times n}$ is given by,

$$A^T = [a_{ji}]_{n\times m}$$

Problem. Create another matrix which is the transpose of a given matrix.

Code.

```
def transpose(matrix):
  rows=len(matrix)
  columns=len(matrix[0])
  transpose_matrix=null(columns, rows) #null matrix
  for i in range(rows):
    for j in range(columns):
      transpose_matrix[j][i]+=matrix[i][j] #definition of transpose

  return transpose_matrix
```

If the matrix is $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 1 & 1 \end{bmatrix}$

, the output is,

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

1.5 Addition of Two Matrices

Given two matrices $A = [a_{ij}]_{n_1\times m_1}$ and $B = [b_{ij}]_{n_2\times m_2}$, the sum $A + B$ is defined only if $(n_1, m_1) = (n_2, m_2) = (n, m)$ (Say), i.e. the matrices have the same order.

$$C = A + B = [a_{ij} + b_{ij}]_{n\times m}$$

Problem. Find the sum of two matrices.

Code. It returns the sum of the matrices if it exists, else, returns -1.

```
def Matrix_Sum(matrix1,matrix2):
    if len(matrix1)==len(matrix2) and len(matrix1[0])==len(matrix2[0]): #Checking for compatibility
    for addition
        rows, columns=len(matrix1), len(matrix1[0])
        sum_matrix=null(rows,columns) #null matrix
        for i in range(rows):
            for j in range(columns):
                sum_matrix[i][j]+=matrix1[i][j]+matrix2[i][j] #definition of sum of matrices
        return sum_matrix
    else:
        return -1
```

If the input matrices are $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 1 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 2 & 2 & 2 \end{bmatrix}$, the output is,

```
[[2, 4, 6], [8, 10, 12], [3, 3, 3]]
```

If the matrices are $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 1 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, the output is,

-1

1.6 Subtraction of Two Matrices

Matrices also have am additive inverse, i.e., there exists a matrix B such that,

$$A + B = 0 \implies B = -A$$

The matrix B is defined as $B = [-a_{ij}]_{n\times m}$ if the matrix A js defined as $A = [a_{ij}]_{n\times m}$.
Thus, we can define the operation $A - B$ as $A + (-B)$ which is simple matrix addition.

Problem. Find the difference of two matrices.

Code. It is similar to the sum of two matrices program.

```
def Matrix_Difference(matrix1,matrix2):
#The program is matrix1-matrix2, since subtraction is not commutative
matrix2_new=null(len(matrix2),len(matrix2[0]))
for i in range(len(matrix2)):
  for j in range(len(matrix2[0])):
    matrix2_new[i][j]=-matrix2[i][j] #From the definition of the negative of a matrix
return(Matrix_Sum(matrix1,matrix2_new))#From the definition of difference of two matrices
```

If the input matrices are $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 1 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 2 & 2 & 2 \end{bmatrix}$, the output is,

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

2 Division of a Matrix by a Scalar

When a matrix is divided by a scalar, each element of the matrix is reduced by a factor of the scalar, i.e.,

$$\frac{A}{k} = \left[\frac{a_{ij}}{k}\right]$$

Problem. Find the result of a matrix divided by a scalar.

Code. Using the above definition,

```
def Division(mx, scalar: int):
    new_matrix = null(len(mx), len(mx[0]))
    for i in range(len(mx)):
        for j in range(len(mx[0])):
            new_matrix[i][j] += mx[i][j] / scalar
    return new_matrix
```

2.1 Minor Matrix of an Element

The minor matrix of an element is the matrix obtained from removing the i^{th} row and j^{th} column of a matrix.

Problem. Find the minor matrix of a matrix given the location of the matrix,

Code. For this, first we exclude the i^{th} row. Then, we exclude the j^{th} column .

```
def Minor_Matrix(mx,r,c):
    return [row[:c]+row[c+1:] for row in (mx[:r]+mx[r+1:])]
```

The following is an incredible one-liner.

2.2 Determinant of a Matrix

The determinant is a scalar value that is a function of the entries of a square matrix. It is denoted by $det(A)$, $|A|$ or Δ.
If $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ then, $\Delta = a_{11}a_{22} - a_{21}a_{12}$.

If $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$ then, $\Delta = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} + a_{22}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{33}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$
Before finding the determinant of a matrix, let us look at Cofactors.

2.2.1 Cofactors

The cofactor of an element a_{ij} is denoted by A_{ij} and is given by,

$$A_{ij} = (-1)^{i+j} M_{ij}$$

Where, M_{ij} is the determinant of the minor matrix of the element a_{ij}. The determinant of a matrix is thus given by,

$$\Delta = \sum_{j=0}^{n} a_{1j} A_{1j}$$

Thus, the determinant of a matrix is the sum of the elements multiplied with their cofactors.

Problem. Find the determinant of a matrix.

Code. Since we have a closed form for the determinant of a 2×2 matrix, we can reduce all determinants[a] to finding the determinant of a smaller matrix by using cofactors.

```
def Determinant(mx):
  # for the base case when the matrix is 2*2
  if len(mx) == 2:
    return mx[0][0] * mx[1][1] - mx[0][1] * mx[1][0]
  determinant = 0
  # We will only expand along the first row
  for i in range(len(mx)):
    determinant += (
      ((-1) ** i) * mx[0][i] * Determinant(Minor_Matrix(mx, 0, i))
    )  # By definition
  return determinant
```

[a] So, we need to use recursion.

3 Inverse of a Matrix

If A is a square matrix of order m, and there exists another square matrix B of the same order m, such that $AB = BA = I$, then B is called the inverse matrix of A and it is denoted by A^{-1}. In that case A is said to be invertible. The inverse of a matrix if it exists is unique.

3.1 Adjoint of a Matrix

The adjoint of a square matrix $A = [a_{ij}]_{n\times n}$ is defined as the transpose of the matrix $[A_{ij}]_{n\times n}$, i.e.,

$$\text{adj } A = [A_{ij}]_{n\times n}$$

Problem. Find the adjoint of a matrix.

Code.

```
def Adjoint(mx):
    temp_matrix = []  # Empty matrix before the transpose
    for i in range(len(mx)):
        row = []  # Initialise each row
        for j in range(len(mx)):
            row.append(
                ((-1) ** (i + j)) * Determinant(Minor_Matrix(mx, i, j))
            )  # from the definition of adjoint
        temp_matrix.append(row)
    return transpose(temp_matrix)
```

The inverse of a matrix can be calculated by using:

$$A^{-1} = \frac{1}{|A|}\text{adj } A$$

Problem. Find the inverse of a matrix.

Code. Using the above definition,

```
def Inverse(mx):
  if Determinant(mx) == 0:
    return None  # The inverse does not exist in this case
  else:
    return Division(Adjoint(mx), Determinant(mx))  # By definition
```

4 Multiplication of Two Matrices

The product of two matrices is defined if the number of columns of A is equal to the number of rows of B. If $A = [a_{ij}]_{m\times n}$ and $B = [b_{jk}]_{n\times p}$, then the i^{th} row of A is $\begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{bmatrix}$ and the k^{th} column of B is $\begin{bmatrix} b_{1k} \\ b_{2k} \\ \vdots \\ b_{nk} \end{bmatrix}$, then $c_{ik} = \sum_{j=1}^{n} a_{ij}b_{jk}$.
The matrix $C = [c_{ik}]_{m\times p}$ is the product AB. Matrix Multiplication is not commutative but is associative and distributive. Moreover, $AB = 0 \nRightarrow A = 0 \vee B = 0$.
Here, we will take two approaches to find the product of two matrices.

Problem. Find the product of two matrices.

Code. Here, we will use the exact definition used above. We will loop 3 times.

```
def Multiplication(mx1, mx2):
    if len(mx1[0]) != len(mx2):
        return "The matrices are incompatible for multiplication."
    else:
        result = null(len(mx1), len(mx2[0]))
        for i in range(len(mx1)):  # Loop through rows of first matrix
            for j in range(len(mx2[0])):  # Loop through each column of second matrix
                for k in range(len(mx2)):  # Loop through each row of second matrix
                    result[i][j] += mx1[i][k] * mx2[k][j]  # By definition
        return result
```

The following can also be done one line as follows:

```
def Matrix_Multiplication(mx1,mx2):
    return [[sum([mx1[i][k]*mx2[k][j] for k in range(len(mx1[0]))]) for j in range(len(mx2[0]))] for i
     in range(len(mx1))]
```

This method is however very slow, as it loops three times. If the matrices are square matrices of the same order (n), then the program takes $O(n * n * n) = O(n^3)$ time.

4.1 Strassen Algorithm

Strassen Algorithm[3] is an algorithm for multiplication of two matrices. It is much faster than the traditional multiplication algorithms (like the one mentioned above). However, it only works for square matrices of the order 2^n, where $n \in \mathrm{N}$.

4.1.1 Joining Matrices

Problem. Join any two matrices horizontally.

[3] named after the German mathematician Volker Strassen.

Code.

```
def joining_horizontally(a: list, b: list) -> list[list]:
  n = len(a)
  new_matrix = null(
    len(a), len(a[0]) + len(b[0])
  )  # Null matrix of the same order as the expected output
  for i in range(n):
    for j in range(n):
      new_matrix[i][j] = a[i][j]  # First matrix
      new_matrix[i][j + n] = b[i][j]  # Second matrix
  return new_matrix
```

If the input matrices are $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ and $B = \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}$, then the output matrix is, $\begin{bmatrix} 1 & 2 & 3 & 10 & 11 & 12 \\ 4 & 5 & 6 & 13 & 14 & 15 \\ 7 & 8 & 9 & 16 & 17 & 18 \end{bmatrix}$.

Problem. Join any two matrices vertically.

Code.

```
def joining_vertically(a: list[list], b: list[list]) -> list[list]:
    n = len(mx1)
    new_matrix = []  # Empty list
    for i in mx1:
        new_matrix.append(i)  # First the first matrix
    for j in mx2:
        new_matrix.append(j)  # Now the second matrix
    return new_matrix
```

If the input matrices are $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ and $B = \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}$, then the output matrix is, $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}$.

4.2 Splitting a Matrix into 4 Smaller Matrices

Problem. Split a given matrix into 4 smaller matrices, given the order of the matrix is of the form $2^n \times 2^n$.

Code. We use simple list slicing to achieve the desired output.

```
def split(mx: list[list]) -> list[list]:
  if len(mx) == len(mx[0]) and pow_2(len(mx)):
    n = len(mx) // 2
    a = mx[:n]
    b = mx[n:]
    a_11 = [a[i][:n] for i in range(n)]
    a_12 = [a[i][n:] for i in range(n)]
    a_13 = [b[i][:n] for i in range(n)]
    a_14 = [b[i][n:] for i in range(n)]
  return a_11, a_12, a_13, a_14
```

. Product of two 2×2 matrices.

. Let the matrices be:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \wedge \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Then the product matrix is given, by,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Where, the elements are,

$$C_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$C_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$C_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$C_{22} = a_{21}b_{12} + a_{22}b_{22}$$

However, this uses 8 multiplications. Strassen found a way to use only 7 multiplications. This is a bit faster as additions are faster than multiplications. For this, we calculate the following 7 products:

$$P_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$P_2 = a_{22}(b_{21} - b_{11})$$
$$P_3 = (a_{11} + a_{12})b_{22}$$
$$P_4 = (a_{12} - a_{22})(b_{21} + b_{22})$$
$$P_5 = a_{11}(b_{12} - b_{22})$$
$$P_6 = (a_{21} + a_{22})b_{11}$$
$$P_7 = (a_{11} - a_{21})(b_{11} + b_{12})$$

Now, calculating the elements,

$$C_{11} = P_1 + P_4 - P_5 - P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_6$$
$$C_{22} = P_1 + P_5 - P_6 - P_7$$

Now, for a bigger matrix, i.e. of the order n, the elements a_{ij} will be matrices and the product will be calculated recursively.
For the time complexity of the previous method,

We reduce the order of the matrix to $\frac{n}{2}$ in each step and calculate 8 products and we add n^2 times, thus,

$$T(n) = 8T(\frac{n}{2}) + n^2$$

On solving the recurrence relation, we get,

$$T(n) = O(n^3)$$

Which is the same as using 3 loops.
Now, for the time complexity of Strassen Algorithm. Similarly,

$$T(n) = 7T(\frac{n}{2}) + n^2$$

Here, we get,

$$T(n) = O(n^{\log_2 7}) = O(2^{2.81})$$

Which is clearly faster than the above mentioned methods! □

Problem. Using the Strassen Algorithm.

Code. It works on the principle of divide and conquer.

```
def strassen(mx1: list[list], mx2: list[list]) -> list[list]:
  if len(mx1) == 1:
    return [[mx1[0][0] * mx2[0][0]]]
  a_11, a_12, a_21, a_22 = split(mx1)
  b_11, b_12, b_21, b_22 = split(mx2)
  p_1, p_2, p_3, p_4, p_5, p_6, p_7 = (
    strassen(Matrix_Sum(a_11, a_22), Matrix_Sum(b_11, b_22)),
    strassen(a_22, Matrix_Difference(b_21, b_11)),
    strassen(Matrix_Sum(a_11, a_12), b_22),
    strassen(Matrix_Difference(a_12, a_22), Matrix_Sum(b_21, b_22)),
    strassen(a_11, Matrix_Difference(b_12, b_22)),
```

```
    strassen(Matrix_Sum(a_21, a_22), b_11),
    strassen(Matrix_Difference(a_11, a_21), Matrix_Sum(b_11, b_12)),
  )  # Exactly from the theory
  c_11, c_12, c_21, c_22 = (
    Matrix_Difference(Matrix_Sum(Matrix_Sum(p_1, p_2), p_4), p_3),
    Matrix_Sum(p_3, p_5),
    Matrix_Sum(p_2, p_6),
    Matrix_Difference(Matrix_Sum(p_1, p_5), Matrix_Sum(p_6, p_7)),
  )  # Exactly from the theory
  return joining_vertically(
    joining_horizontally(c_11, c_12), joining_horizontally(c_21, c_22)
  )
```