PROGRAM 10:LINEAR ALGEBRA USING NUMPY

Requirement:

Create 2 random square matrices and perform matrix addition, matrix multiplication, transpose, determinant and inverse of a matrix without using inbuilt function. Validate your result with the help of built in function both the output should be same.

INDEX:

In [2]:

```
import numpy as np
from scipy import linalg
```

1. Two Random Square Matrices

[Integer Values]

In [3]:

```
rand_matrix1 = np.random.randint(10, size=(3, 3))
np.matrix(rand_matrix1)
```

Out[3]:

```
matrix([[9, 9, 5],
        [4, 6, 1],
        [8, 8, 9]])
```

In [4]:

```
rand_matrix2 = np.random.randint(10, size=(3, 3))
np.matrix(rand_matrix2)
```

Out[4]:

```
matrix([[9, 4, 7],
        [4, 6, 9],
        [9, 4, 0]])
```

In [5]:

```
rand_matrix3 = np.random.randint(10, size=(2, 2))
np.matrix(rand_matrix3)
```

Out[5]:

```
matrix([[6, 3],
        [0, 9]])
```

1.1. Adding Matrices

In [6]:

```
# without built-in function

M1_Add  = np.zeros((3, 3))

for i in range(len(rand_matrix1)):
    for k in range(len(rand_matrix2)):
        M1_Add[i][k] = rand_matrix1[i][k] + rand_matrix2[i][k]
```

In [7]:

```
np.matrix(M1_Add)
```

Out[7]:

```
matrix([[18., 13., 12.],
        [ 8., 12., 10.],
        [17., 12.,  9.]])
```

In [8]:

```
# built-in function

M1_Add1 = np.add(rand_matrix1,rand_matrix2)
np.matrix(M1_Add1)
```

Out[8]:

```
matrix([[18, 13, 12],
        [ 8, 12, 10],
        [17, 12,  9]])
```

1.2. Matrix Multiplication

[element wise matrix multiplication]

In [9]:

```
# without built-in function

M1_mul= np.zeros((3, 3))

for i in range(len(rand_matrix1)):
    for k in range(len(rand_matrix2)):
        M1_mul[i][k] = rand_matrix1[i][k] * rand_matrix2[i][k]
```

In [10]:

```
np.matrix(M1_mul)
```

Out[10]:

```
matrix([[81., 36., 35.],
        [16., 36.,  9.],
        [72., 32.,  0.]])
```

In [11]:

```
# built-in function

M1_mul1 = np.multiply(rand_matrix1,rand_matrix2)
np.matrix(M1_mul1)
```

Out[11]:

```
matrix([[81, 36, 35],
        [16, 36,  9],
        [72, 32,  0]])
```

[matrix product of two arrays]

In [12]:

```
# without built-in function

def Matrix_mul(a,b):
    c = []
    for i in range(0,len(a)):
        temp=[]
        for j in range(0,len(b[0])):
            s = 0
            for k in range(0,len(a[0])):
                s += a[i][k]*b[k][j]
            temp.append(s)
        c.append(temp)
    return c
```

In [13]:

```
k=Matrix_mul(rand_matrix1, rand_matrix2)
np.matrix(k)
```

Out[13]:

```
matrix([[162, 110, 144],
        [ 69,  56,  82],
        [185, 116, 128]])
```

In [14]:

```
# built-in function

M2_mul2 = np.matmul(rand_matrix1,rand_matrix2)
np.matrix(M2_mul2)
```

Out[14]:

```
matrix([[162, 110, 144],
        [ 69,  56,  82],
        [185, 116, 128]])
```

[dot product of two arrays]

In [15]:

```
# without built-in function

def Matrix_mul(a,b):
    c = []
    for i in range(0,len(a)):
        temp=[]
        for j in range(0,len(b[0])):
            s = 0
            for k in range(0,len(a[0])):
                s += a[i][k]*b[k][j]
            temp.append(s)
        c.append(temp)
    return c
```

In [16]:

```
k=Matrix_mul(rand_matrix1, rand_matrix2)
np.matrix(k)
```

Out[16]:

```
matrix([[162, 110, 144],
        [ 69,  56,  82],
        [185, 116, 128]])
```

In [17]:

```
# built-in function

M2_mul3 = rand_matrix1.dot(rand_matrix2)
M2_mul3
```

Out[17]:

```
array([[162, 110, 144],
       [ 69,  56,  82],
       [185, 116, 128]])
```

1.3. Transpose of a Matrix

In [18]:

```
# without built-in function

def Mat_Trans(X):
    M2_Trans= np.zeros((3, 3))

    for i in range(len(X)):
        for j in range(len(X[0])):
            M2_Trans[j][i] = X[i][j]

    return M2_Trans
```

In [19]:

```
m=Mat_Trans(rand_matrix1)
np.matrix(m)
```

Out[19]:

```
matrix([[9., 4., 8.],
        [9., 6., 8.],
        [5., 1., 9.]])
```

In [20]:

```
# built-in function

m1=rand_matrix1.transpose()
np.matrix(m1)
```

Out[20]:

```
matrix([[9, 4, 8],
        [9, 6, 8],
        [5, 1, 9]])
```

1.4. Determinant of a Matrix

In [21]:

```
# without built-in function

def determinant(a):
    assert len(a.shape) == 2
    assert a.shape[0] == a.shape[1]
    n = a.shape[0]

    for k in range(0, n-1):

        for i in range(k+1, n):
            if a[i,k] != 0.0:
                lam = a [i,k]/a[k,k]
                a[i,k:n] = a[i,k:n] - lam*a[k,k:n]

    return np.prod(np.diag(a))
```

In [22]:

```
determinant(rand_matrix1)
```

Out[22]:

```
72
```

In [23]:

```
# built-in function

np.linalg.det(rand_matrix1)
```

Out[23]:

```
72.0
```

1.5. Inverse of a Matrix

In [24]:

```
# without built-in function

def getMatrixMinor(m,i,j):
    return [row[:j] + row[j+1:] for row in (m[:i]+m[i+1:])]

def getMatrixInverse(m):
    determinant = np.linalg.det(m)
    #special case for 2x2 matrix:
    if len(m) == 2:
        return [[m[1][1]/determinant, -1*m[0][1]/determinant],
                [-1*m[1][0]/determinant, m[0][0]/determinant]]

    #find matrix of cofactors
    cofactors = []
    for r in range(len(m)):
        cofactorRow = []
        for c in range(len(m)):
            minor = getMatrixMinor(m,r,c)
            cofactorRow.append(((-1)**(r+c)) * getMatrixDeternminant(minor))
        cofactors.append(cofactorRow)
    cofactors = cofactors.transpose()
    for r in range(len(cofactors)):
        for c in range(len(cofactors)):
            cofactors[r][c] = cofactors[r][c]/determinant
    return cofactors
```

In [25]:

```
m3=getMatrixInverse(rand_matrix3)
np.matrix(m3)
```

Out[25]:

```
matrix([[ 0.16666667, -0.05555556],
        [ 0.        ,  0.11111111]])
```

In [26]:

```
# built-in function

m4=np.linalg.inv(rand_matrix3)
np.matrix(m4)
```

Out[26]:

```
matrix([[ 0.16666667, -0.05555556],
        [ 0.        ,  0.11111111]])
```

n x n matrix

In [27]:

```
def inversematrix(M1,s):
    M=np.copy(M1)
    I=np.eye(s,dtype='float64')
    M=M.astype('float64')

    for i in range(0,s):
        a1=M[i,i]
        I[i]=I[i]/a1
        M[i]=M[i]/a1
        if i==0:
            for j in range(i+1,s):
                a2=M[j,i]
                I[j]=I[j]-I[i]*a2
                M[j]=M[j]-M[i]*a2
        else:
            L=list(range(0,s))
            L.remove(i)
            for j in L:
                a2=M[j,i]
                I[j]=I[j]-I[i]*a2
                M[j]=M[j]-M[i]*a2
    return I
```

In [28]:

```
# without built-in function
s=rand_matrix1.shape
inv=inversematrix(rand_matrix1,s[0])
np.matrix(inv)
```

Out[28]:

```
matrix([[ 0.11111111, -0.5       , -0.26388889],
        [ 0.        ,  0.5       ,  0.125     ],
        [ 0.        ,  0.        ,  0.25      ]])
```

In [29]:

```
# built-in function
inv1=np.linalg.inv(rand_matrix1)
np.matrix(inv1)
```

Out[29]:

```
matrix([[ 0.11111111, -0.5       , -0.26388889],
        [ 0.        ,  0.5       ,  0.125     ],
        [ 0.        ,  0.        ,  0.25      ]])
```

2. Two Random Square Matrices

[Floating Values]

In [30]:

```
random_matrix1 = np.random.rand(3, 3)
np.matrix(random_matrix1)
```

Out[30]:

```
matrix([[0.75122206, 0.60382605, 0.00111868],
        [0.35650984, 0.93741511, 0.99686101],
        [0.31066982, 0.0255548 , 0.74108756]])
```

In [31]:

```
random_matrix2 = np.random.rand(3, 3)
np.matrix(random_matrix2)
```

Out[31]:

```
matrix([[0.86753739, 0.02138478, 0.10084754],
        [0.9012808 , 0.41652693, 0.27279477],
        [0.76776928, 0.87033751, 0.74405778]])
```

In [32]:

```
random_matrix3 = np.random.rand(2, 2)
np.matrix(random_matrix3)
```

Out[32]:

```
matrix([[0.27510424, 0.93309674],
        [0.02523341, 0.64379028]])
```

2.1. Adding Matrices

In [33]:

```
# without built-in function

def Mat_Add(A,B):
    M2_Add= np.zeros((3, 3))

    for i in range(len(random_matrix1)):
        for j in range(len(random_matrix2)):
            M2_Add[i][j] = random_matrix1[i][j] + random_matrix2[i][j]

    return M2_Add
```

In [34]:

```
M1=Mat_Add(random_matrix1,random_matrix2)
np.matrix(M1)
```

Out[34]:

```
matrix([[1.61875945, 0.62521083, 0.10196622],
        [1.25779064, 1.35394204, 1.26965578],
        [1.0784391 , 0.89589231, 1.48514534]])
```

In [35]:

```
# built-in function

M1_Add1 = np.add(random_matrix1,random_matrix2)
np.matrix(M1_Add1)
```

Out[35]:

```
matrix([[1.61875945, 0.62521083, 0.10196622],
        [1.25779064, 1.35394204, 1.26965578],
        [1.0784391 , 0.89589231, 1.48514534]])
```

2.2. Matrix Multiplication

[element wise matrix multiplication]

In [36]:

```
# without built-in function

M2_mul= np.zeros((3, 3))

for i in range(len(random_matrix1)):
    for k in range(len(random_matrix2)):
        M2_mul[i][k] = random_matrix1[i][k] * random_matrix2[i][k]

np.matrix(M2_mul)
```

Out[36]:

```
matrix([[6.51713225e-01, 1.29126857e-02, 1.12816002e-04],
        [3.21315477e-01, 3.90458640e-01, 2.71938474e-01],
        [2.38522745e-01, 2.22413005e-02, 5.51411966e-01]])
```

In [37]:

```
# built-in function

M2_mul1 = np.multiply(random_matrix1,random_matrix2)
np.matrix(M2_mul1)
```

Out[37]:

```
matrix([[6.51713225e-01, 1.29126857e-02, 1.12816002e-04],
        [3.21315477e-01, 3.90458640e-01, 2.71938474e-01],
        [2.38522745e-01, 2.22413005e-02, 5.51411966e-01]])
```

[matrix product of two arrays]

In [38]:

```
def Matrix_mul(a,b):
    c = []
    for i in range(0,len(a)):
        temp=[]
        for j in range(0,len(b[0])):
            s = 0
            for k in range(0,len(a[0])):
                s += a[i][k]*b[k][j]
            temp.append(s)
        c.append(temp)
    return c
```

In [39]:

```
M2=Matrix_mul(random_matrix1, random_matrix2)
np.matrix(M2)
```

Out[39]:

```
matrix([[1.19678894, 0.26854816, 0.24131185],
        [1.91951912, 1.26568805, 1.03339728],
        [0.861534  , 0.66228417, 0.58971347]])
```

In [40]:

```
# built-in function

M2_mul2 = np.matmul(random_matrix1,random_matrix2)
np.matrix(M2_mul2)
```

Out[40]:

```
matrix([[1.19678894, 0.26854816, 0.24131185],
        [1.91951912, 1.26568805, 1.03339728],
        [0.861534  , 0.66228417, 0.58971347]])
```

[dot product of two arrays]

In [41]:

```
def Matrix_mul(a,b):
    c = []
    for i in range(0,len(a)):
        temp=[]
        for j in range(0,len(b[0])):
            s = 0
            for k in range(0,len(a[0])):
                s += a[i][k]*b[k][j]
            temp.append(s)
        c.append(temp)
    return c
```

In [42]:

```
M3=Matrix_mul(random_matrix1, random_matrix2)
np.matrix(M3)
```

Out[42]:

```
matrix([[1.19678894, 0.26854816, 0.24131185],
        [1.91951912, 1.26568805, 1.03339728],
        [0.861534  , 0.66228417, 0.58971347]])
```

In [43]:

```
# built-in function

M2_mul3 = random_matrix1.dot(random_matrix2)
np.matrix(M2_mul3)
```

Out[43]:

```
matrix([[1.19678894, 0.26854816, 0.24131185],
        [1.91951912, 1.26568805, 1.03339728],
        [0.861534  , 0.66228417, 0.58971347]])
```

2.3. Transpose of a Matrix

In [44]:

```
# without built-in function

def Mat_Trans(X):
    M2_Trans= np.zeros((3, 3))

    for i in range(len(X)):
        for j in range(len(X[0])):
            M2_Trans[j][i] = X[i][j]

    return M2_Trans
```

In [45]:

```
M4=Mat_Trans(random_matrix1)
np.matrix(M4)
```

Out[45]:

```
matrix([[0.75122206, 0.35650984, 0.31066982],
        [0.60382605, 0.93741511, 0.0255548 ],
        [0.00111868, 0.99686101, 0.74108756]])
```

In [46]:

```
# built-in function

M5=random_matrix1.transpose()
np.matrix(M5)
```

Out[46]:

```
matrix([[0.75122206, 0.35650984, 0.31066982],
        [0.60382605, 0.93741511, 0.0255548 ],
        [0.00111868, 0.99686101, 0.74108756]])
```

2.4. Determinant of a Matrix

In [47]:

```
# without built-in function

def determinant(a):
    assert len(a.shape) == 2
    assert a.shape[0] == a.shape[1]
    n = a.shape[0]

    for k in range(0, n-1):

        for i in range(k+1, n):
            if a[i,k] != 0.0:
                lam = a [i,k]/a[k,k]
                a[i,k:n] = a[i,k:n] - lam*a[k,k:n]

    return np.prod(np.diag(a))
```

In [48]:

```
determinant(random_matrix1)
```

Out[48]:

```
0.5298941357320285
```

In [49]:

```
# built-in function

np.linalg.det(random_matrix1)
```

Out[49]:

```
0.5298941357320285
```

2.5. Inverse of a Matrix

[2 x 2 Matrix]

In [50]:

```
# without built-in function

def getMatrixMinor(m,i,j):
    return [row[:j] + row[j+1:] for row in (m[:i]+m[i+1:])]

def getMatrixInverse(m):
    determinant = np.linalg.det(m)
    if len(m) == 2:
        return [[m[1][1]/determinant, -1*m[0][1]/determinant],
                [-1*m[1][0]/determinant, m[0][0]/determinant]]

    #find matrix of cofactors
    cofactors = []
    for r in range(len(m)):
        cofactorRow = []
        for c in range(len(m)):
            minor = getMatrixMinor(m,r,c)
            cofactorRow.append(((-1)**(r+c)) * getMatrixDeternminant(minor))
        cofactors.append(cofactorRow)
    cofactors = cofactors.transpose()
    for r in range(len(cofactors)):
        for c in range(len(cofactors)):
            cofactors[r][c] = cofactors[r][c]/determinant
    return cofactors
```

In [51]:

```
M6=getMatrixInverse(random_matrix3)
np.matrix(M6)
```

Out[51]:

```
matrix([[ 4.19231957, -6.07626403],
        [-0.1643183 ,  1.79146052]])
```

In [52]:

```
# built-in function

M7=np.linalg.inv(random_matrix3)
np.matrix(M7)
```

Out[52]:

```
matrix([[ 4.19231957, -6.07626403],
        [-0.1643183 ,  1.79146052]])
```

n x n matrix

In [53]:

```
def inversematrix(M1,s):
    M=np.copy(M1)
    I=np.eye(s,dtype='float64')
    M=M.astype('float64')

    for i in range(0,s):
        a1=M[i,i]
        I[i]=I[i]/a1
        M[i]=M[i]/a1
        if i==0:
            for j in range(i+1,s):
                a2=M[j,i]
                I[j]=I[j]-I[i]*a2
                M[j]=M[j]-M[i]*a2
        else:
            L=list(range(0,s))
            L.remove(i)
            for j in L:
                a2=M[j,i]
                I[j]=I[j]-I[i]*a2
                M[j]=M[j]-M[i]*a2
    return I
```

In [54]:

```
# without built-in function
s=random_matrix1.shape
inv=inversematrix(random_matrix1,s[0])
np.matrix(inv)
```

Out[54]:

```
matrix([[ 1.33116431, -1.23497725,  1.13396609],
        [ 0.        ,  1.53643944, -1.41248055],
        [ 0.        ,  0.        ,  0.92270691]])
```

In [55]:

```
# built-in function
inv1=np.linalg.inv(random_matrix1)
np.matrix(inv1)
```

Out[55]:

```
matrix([[ 1.33116431, -1.23497725,  1.13396609],
        [ 0.        ,  1.53643944, -1.41248055],
        [ 0.        ,  0.        ,  0.92270691]])
```