Numpy (Demo)

In [1]:
```
import numpy as np
```

1. Creation

In [2]:
```
# Create a numpy matrix from a list
# elements' type must be homogeneous
A = np.array([[1, 2, 3], [4, 5, 6]])

A
```

Out[2]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [3]:
```
# Create an identity matrix (unit matrix): its diagonal elements equal to 1, and zeroes everywhere else.
I = np.eye(5)

I
```

Out[3]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [4]:
```
# Create an diagonal matrix from the first element
I2 = np.eye(5, 3)

I2
```

Out[4]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [5]:
```
# Create a vector of 5 elements having a value of 1
O = np.ones(5)

O
```

Out[5]:
```
array([1., 1., 1., 1., 1.])
```

In [6]:
```
# Create a 3X5 matrix of 1's
O2 = np.ones((3, 5))

O2
```

Out[6]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

In [7]:
```
# Create a 3X5 matrix of 0's
Z = np.zeros((3, 5))

Z
```

Out[7]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [8]:
```
# Create a 3X5 matrix of values 2.2 (a given value)
V = np.full((3, 5), 2.2)

V
```

Out[8]:
```
array([[2.2, 2.2, 2.2, 2.2, 2.2],
       [2.2, 2.2, 2.2, 2.2, 2.2],
       [2.2, 2.2, 2.2, 2.2, 2.2]])
```

In [9]:
```
# Create a vector of values in the range [0, 5[ with a step of 1
C = np.arange(0, 5)

C
```

Out[9]:
```
array([0, 1, 2, 3, 4])
```

In [10]:
```python
# Create a vector of values in the range [0, 5[ with a step of 0.5
P = np.arange(0, 5, 0.5)

P
```

Out[10]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

In [11]:
```python
# Create a 3X5 matrix with random values in [0, 1]
A  = np.random.rand(3, 5)

A
```

Out[11]:
```
array([[0.54096929, 0.50299421, 0.33775669, 0.42727139, 0.6163162 ],
       [0.84328727, 0.81019213, 0.60852146, 0.77315708, 0.7967893 ],
       [0.25197899, 0.48297844, 0.08413812, 0.6285931 , 0.66382411]])
```

In [12]:
```python
# Create a vector of 11 elements with values in [0, 5]
# The same as np.arange(0, 5.000000000001, (5-0+1)/11)
# 5.000000001 instead of 5. is used to include 5 in the vector
B = np.linspace(0, 5, 11)

B
```

Out[12]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

In [13]:
```python
# Create a diagonal matrix with a given value
D = np.diag([5, 2, 3, 5])

D
```

Out[13]:
```
array([[5, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 5]])
```

In [14]:
```python
# Create a diagonal matrix with a given vector and a shift from the first element
D1 = np.diag([5, 2, 3, 5], 1)

D1
```

Out[14]:
```
array([[0, 5, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 5],
       [0, 0, 0, 0, 0]])
```

In [15]:
```
# Create a vandermonde matrix from a vector T
# The vector will be considered as a column
# The last column is T^0, the one before is T^1, then T^2, etc.
V = np.vander([2, 3, 5], 3)
V
```

Out[15]:
```
array([[ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

In [16]:
```
A = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

# When referencing a part of a matrix, we haven't actually created a new matrix
# It's just a reference to a portion of the original matrix
# To create a new matrix, we use the copy() method
B = A[:, :-1].copy()

B
```

Out[16]:
```
array([[1, 2],
       [4, 5]])
```

2. Transformation

In [17]:
```
# We are going to use two matrices for the upcoming transformations
A = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

B = np.array([
```

```
    [7, 8, 9],
    [10, 11, 11]
])

A, B
```

Out[17]:
```
(array([[1, 2, 3],
        [4, 5, 6]]),
 array([[ 7,  8,  9],
        [10, 11, 11]]))
```

In [18]:
```
# Concatenate the two matrices along axis 0 (the rows)
# In this case, the number of columns in both matrices must be identical
ABv = np.concatenate((A, B), axis=0)

ABv
```

Out[18]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 11]])
```

In [19]:
```
# Concatenate the two matrices along axis 1 (the columns)
# In this case, the number of rows in both matrices must be identical
ABh = np.concatenate((A, B), axis=1)

ABh
```

Out[19]:
```
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 11]])
```

3. Indexing

In [20]:
```
A = np.array([
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 0]
])

A
```

Out[20]:
```
array([[1, 2, 3, 4, 5],
       [6, 7, 8, 9, 0]])
```

In [21]:
```
# Retrieve the shape of the matrix: the dimensions and the number of elements
# The shape is a tuple
# The number of elements in the tuple indicates the number of dimensions
# This matrix has two dimensions: the first with 2 elements and the second with 5

A.shape
```

Out[21]:
```
(2, 5)
```

In [22]:
```
# Let's take an example of a matrix with 3 dimensions
A3 = np.array([
    [[1, 2, 5, 5], [3, 4, 5, 5]],
    [[6, 7, 7, 8], [8, 9, 8, 5]],
    [[6, 7, 7, 8], [8, 9, 8, 5]]
])

A3
```

Out[22]:
```
array([[[1, 2, 5, 5],
        [3, 4, 5, 5]],

       [[6, 7, 7, 8],
        [8, 9, 8, 5]],

       [[6, 7, 7, 8],
        [8, 9, 8, 5]]])
```

In [23]:
```
# The dimension is 3X2X4
A3.shape
```

Out[23]:
```
(3, 2, 4)
```

In [24]:
```
# Retrieve the number of elements in the third dimension
A3.shape[2]
```

Out[24]:
```
4
```

In [25]:
```
A
```

Out[25]:
```
array([[1, 2, 3, 4, 5],
       [6, 7, 8, 9, 0]])
```

In [26]:
```
# Get a portion of the matrix: all rows, columns starting from the 2nd
A[:, 1:]
```

Out[26]:
```
array([[2, 3, 4, 5],
       [7, 8, 9, 0]])
```

In [27]:
```
# Get a portion of the matrix: all rows, all columns except the last one
A[:, :-1]
```

Out[27]:
```
array([[1, 2, 3, 4],
       [6, 7, 8, 9]])
```

In [28]:
```
# Get a portion of the matrix: all rows, last column
A[:, -1:]
```

Out[28]:
```
array([[5],
       [0]])
```

In [29]:
```
# Get a portion of the matrix: all rows, last 2 columns
A[:, -2:]
```

Out[29]:
```
array([[4, 5],
       [9, 0]])
```

In [30]:
```
# Retrieve elements using a mask
# Here, we want to keep all the rows
# For the columns, we only want to keep the first and the third
mask = [True, False, True, False, False]

A[:, mask]
```

Out[30]:
```
array([[1, 3],
       [6, 8]])
```

In [31]:
```
# The same as before, but by specifying the index of the column to keep
idx = [0, 2]
```

```
A[:, idx]
```

Out[31]:

```
array([[1, 3],
       [6, 8]])
```

In [32]:

```
A.shape
```

Out[32]:

```
(2, 5)
```

In [33]:

```
# Aaa a dimension into the matrix
B = A[:, np.newaxis, :]

B
```

Out[33]:

```
array([[[1, 2, 3, 4, 5]],

       [[6, 7, 8, 9, 0]]])
```

In [34]:

```
B.shape
```

Out[34]:

```
(2, 1, 5)
```

In [35]:

```
A = np.array([
    [2, 5, 6, 8, 4],
    [1, 2, 7, 9, 2],
    [2, 3, 7, 8, 9]
])

# Sort the rows (axis=0) of a matrix according to the first column (if tied, according to the second, and s
np.sort(A, axis=0)
```

Out[35]:

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

4. Search

In [36]:

```
A
```

Out[36]:
```
array([[2, 5, 6, 8, 4],
       [1, 2, 7, 9, 2],
       [2, 3, 7, 8, 9]])
```

In [37]:
```
# Trouver l'indice de l'élément max dans les lignes
# ça va retourner un vecteur d'une taille égale au nombre des colonnes
# chaque élément représente l'indice de la ligne contenant la valeur max
np.argmax(A, axis=0)
```

Out[37]:
```
array([0, 0, 1, 1, 2])
```

In [38]:
```
# Même chose, mais pour les colonnes
np.argmax(A, axis=1)
```

Out[38]:
```
array([3, 3, 4])
```

In [39]:
```
# Retourne l'indice de l'élément max en considérant la matrice comme étant un vecteur
np.argmax(A)
```

Out[39]:
```
8
```

In [40]:
```
# Retourner les indices des éléments qui satisfont une condition donnée
np.argwhere(A > 5)
```

Out[40]:
```
array([[0, 2],
       [0, 3],
       [1, 2],
       [1, 3],
       [2, 2],
       [2, 3],
       [2, 4]])
```

5. Opérations

In [41]:
```
A
```

Out[41]:
```
array([[2, 5, 6, 8, 4],
       [1, 2, 7, 9, 2],
       [2, 3, 7, 8, 9]])
```

In [42]:
```
# Transpose of a matrix
A.T
```

Out[42]:
```
array([[2, 1, 2],
       [5, 2, 3],
       [6, 7, 7],
       [8, 9, 8],
       [4, 2, 9]])
```

In [43]:
```
# The exponent of a matrix: element by element (element-wise)
A**2
```

Out[43]:
```
array([[ 4, 25, 36, 64, 16],
       [ 1,  4, 49, 81,  4],
       [ 4,  9, 49, 64, 81]])
```

In [44]:
```
# Sum of two matrices: element by element
# Both matrices must have the same shape
A + A
```

Out[44]:
```
array([[ 4, 10, 12, 16,  8],
       [ 2,  4, 14, 18,  4],
       [ 4,  6, 14, 16, 18]])
```

In [45]:
```
# Multiplication of a scalar by a matrix:
# Each element of the matrix will be multiplied by this scalar
2 * A
```

Out[45]:
```
array([[ 4, 10, 12, 16,  8],
       [ 2,  4, 14, 18,  4],
       [ 4,  6, 14, 16, 18]])
```

In [46]:
```
# Matrix multiplication between two matrices:
# The number of columns in the first must be equal to the number of rows in the second
B = np.array([
    [5, 2],
    [2, 3],
    [1, 4],
    [2, 2],
    [3, 1]
```

```
])
A @ B
```

Out[46]:
```
array([[54, 63],
       [40, 56],
       [66, 66]])
```

6. Mathematical functions

In [47]:
```
A
```

Out[47]:
```
array([[2, 5, 6, 8, 4],
       [1, 2, 7, 9, 2],
       [2, 3, 7, 8, 9]])
```

In [48]:
```
# Exponential
np.exp(A)
```

Out[48]:
```
array([[7.38905610e+00, 1.48413159e+02, 4.03428793e+02, 2.98095799e+03,
        5.45981500e+01],
       [2.71828183e+00, 7.38905610e+00, 1.09663316e+03, 8.10308393e+03,
        7.38905610e+00],
       [7.38905610e+00, 2.00855369e+01, 1.09663316e+03, 2.98095799e+03,
        8.10308393e+03]])
```

In [49]:
```
# Logarithm
np.log(A)
```

Out[49]:
```
array([[0.69314718, 1.60943791, 1.79175947, 2.07944154, 1.38629436],
       [0.        , 0.69314718, 1.94591015, 2.19722458, 0.69314718],
       [0.69314718, 1.09861229, 1.94591015, 2.07944154, 2.19722458]])
```

In [50]:
```
# Logarithm base 2
np.log2(A)
```

Out[50]:
```
array([[1.        , 2.32192809, 2.5849625 , 3.        , 2.        ],
       [0.        , 1.        , 2.80735492, 3.169925  , 1.        ],
       [1.        , 1.5849625 , 2.80735492, 3.        , 3.169925  ]])
```

In [51]:
```
# Logarithm base 10
```

```
np.log10(A)
```

Out[51]:
```
array([[0.30103   , 0.69897   , 0.77815125, 0.90308999, 0.60205999],
       [0.        , 0.30103   , 0.84509804, 0.95424251, 0.30103   ],
       [0.30103   , 0.47712125, 0.84509804, 0.90308999, 0.95424251]])
```

In [52]:
```
# Square root
np.sqrt(A)
```

Out[52]:
```
array([[1.41421356, 2.23606798, 2.44948974, 2.82842712, 2.        ],
       [1.        , 1.41421356, 2.64575131, 3.        , 1.41421356],
       [1.41421356, 1.73205081, 2.64575131, 2.82842712, 3.        ]])
```

In [53]:
```
A
```

Out[53]:
```
array([[2, 5, 6, 8, 4],
       [1, 2, 7, 9, 2],
       [2, 3, 7, 8, 9]])
```

In [54]:
```
# Sum of all matrix's elements
A.sum()
```

Out[54]:
```
75
```

In [55]:
```
# Sum of all the rows of a matrix
# Gives a vector of size equal to the number of columns
A.sum(axis=0)
```

Out[55]:
```
array([ 5, 10, 20, 25, 15])
```

In [56]:
```
# Sum of all the columns of a matrix
# Gives a vector of size equal to the number of rows
A.sum(axis=1)
```

Out[56]:
```
array([25, 21, 29])
```

In [57]:
```
# Average of all the rows of a matrix
# Gives a vector of size equal to the number of columns
A.mean(axis=0)
```

Out[57]:
```
array([1.66666667, 3.33333333, 6.66666667, 8.33333333, 5.        ])
```

In [58]:
```
# Standard deviation of all the rows of a matrix
# Gives a vector of size equal to the number of columns
A.std(axis=0)
```

Out[58]:
```
array([0.47140452, 1.24721913, 0.47140452, 0.47140452, 2.94392029])
```

In [59]:
```
# Max of all the rows of a matrix
# Gives a vector of size equal to the number of columns
A.max(axis=0)
```

Out[59]:
```
array([2, 5, 7, 9, 9])
```

In [60]:
```
V = np.array([5, 2, 5, 3, 5, 2, 2])

# Retrieve the unique elements
np.unique(V)
```

Out[60]:
```
array([2, 3, 5])
```

In [61]:
```
V = np.array(["C", "A", "B", "A", "B", "A"])
# Retrieve the unique elements and their freuencies
np.unique(V, return_counts=True)
```

Out[61]:
```
(array(['A', 'B', 'C'], dtype='<U1'), array([3, 2, 1]))
```

7. Logical functions

In [62]:
```
A
```

Out[62]:
```
array([[2, 5, 6, 8, 4],
       [1, 2, 7, 9, 2],
       [2, 3, 7, 8, 9]])
```

In [63]:
```
# Apply a logical operation to test the elements of a matrix
B = A > 5

B
```

Out[63]:
```
array([[False, False,  True,  True, False],
       [False, False,  True,  True, False],
       [False, False,  True,  True,  True]])
```

In [64]:
```
# Convert boolean elements to integers
B.astype(int)
```

Out[64]:
```
array([[0, 0, 1, 1, 0],
       [0, 0, 1, 1, 0],
       [0, 0, 1, 1, 1]])
```

In [65]:
```
AA = np.array([False, True, False, True])

BB = np.array([False, False, True, True])
```

In [66]:
```
# Logical AND
AA & BB
```

Out[66]:
```
array([False, False, False,  True])
```

In [67]:
```
# Logical OR
AA | BB
```

Out[67]:
```
array([False,  True,  True,  True])
```

In [68]:
```
# Logical NOT
np.logical_not(AA)
```

Out[68]:
```
array([ True, False,  True, False])
```

In [ ]: