NumPy

1.Introduction to NumPy

```
* NumPy is an Open-source library in Python that provides support for large ,multi-dimentional
arrays and matrices ,along with a collection of mathematical functions to operate on these
arrays.
```

Key Features:

- 1.Efficient Storage :Numpy arrays use less memory comapared to Python Lists.
- 2.Performance:NumPy opearations are implemented in C and optimisation for performance,allowing for faster computations
- 3.Interoperability : Works seamlessly with many other scientific libraries,including #SciPy,#Pandas and #MatplotLib

* Difference Between Array and Matrix

* NumPy arrays(ndarray):

- Difinition: A versitile container for homogenious data types(all elements must be of the same type).
- Dimentionality: Can be Multi-Dimentional(1D,2D,3D,nD).
- Operations : Supports a wide range of mathematical operations and broadcasting.

Example:

In [27]:
```
import numpy as np

arr=np.array([1,2,3,4]) #1D arrray
print(arr)
```

```
[1 2 3 4]
```

* Numpy Matrix:

- Definition: A specialized 2D array designed for linear algebra opeartions.
- Dimentionality: Always 2-dimentional.
- Operations: Supports matrix-specific opearations (like matrix multiplication using * operator) but is less flexible than arrays.

Example:

In [51]:
```
mat=np.matrix([[1,2],[3,4]])
print(mat)
print( " ")
print(" ")
print(mat.shape)
```

```
[[1 2]
 [3 4]]


(2, 2)
```

In [57]:
```
mat=np.matrix([[1,3,2],[3,4,5],[3,4,2]])
print(mat)
print( " ")
print(" ")
print(mat.shape)
```

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


(3, 3)
```

Key Differences:

- Dimentionality: A matrix can be strictly 2D,while an array can be of any dimentions.
- Functionalities: Numpy arrays supports most versatile opearations,while matrices have specialized mathods for linear algebra.
- Multiplication Behaviour: Using * on Matrix performs matrix multiplication,while for an array ,it performs element -wise multiplications.

*Difference Between range and arange

* Range

- Pupose : A Built-in Python function use to generate a sequence of numbers.
- Returns : A range object ,which is an immutable sequence of numbers (similar to a list but more memory efficiancy)
- Usage : Commonly used in for loops to iterate over a sequence.

Example :

In [71]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Parameters:

- range(start ,stop,step)
- start: (defualt is 0) starting point of the sequence.
- stop : (require) the end point (exclusive).
- step: (defulat is 1) the difference between each number.

numpy .arange:

- Purpose : A function provided by the Numpy library to create a Numpy array with evenly spaced values.
- Returns : A Numpy array
- Usage : Useful when working with NumPy for mathematical opearations on arrays

Example:

In [81]:
```
import numpy as np
arr=np.arange(5)
print(arr)
```

```
[0 1 2 3 4]
```

Parameters:

- np.arange(start ,stop,step)
- start: (defualt is 0) starting point of the sequence.
- stop : (require) the end point (exclusive).
- step: (defulat is 1) the difference between each number.

Key Differencce:

- Output Type: range returns a range object,while numpy.arange a Numpy array.
- Usage : range is used for generating sequences in loops,while arange is specialised for creating numerical arrays in NumPy

2. NumPy Arrays (ndarrays)

2.1 Creation of Arrays

*Using np.array(): Convert Python lists or tuples to Numpy arrays

Example:

In [93]:
```
import numpy as np
list_a=[1,2,3,4]
arr=np.array(list_a)
print(arr)
```

```
[1 2 3 4]
```

$Using np.zeros(),no.ones(),np.empty():

- np.zeros(shape): Creates an array filled with zeros.

- np.ones(shape) : Creates an array filled with ones.
- np.empty(shape): Creates an array without initialising its values(values will be random)

Example :

In [71]:
```
zeros=np.zeros((2,3)) # 2 x 3 array of zeros
print(zeros)

ones=np.ones((2,3))# 2 x 3 array of ones
print(ones)

empty=np.empty((2,2)) # 2 x 2 array (values uninitialised)
print(empty)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]]
[[0.0e+000 4.9e-324]
 [9.9e-324 1.5e-323]]
```

• Using np.arange() and np.linspace():

- np.arange(start,stop,step) : Similar to Python's range () but returns an array
- np.linspace(start,stop,num): Generates evenly spaced values between values start and stop.

Example:

In [147...
```
range_array=np.arange(0,12,2)
print(range_array)

linspace_array=np.linspace(0,1,5)
print(linspace_array)
```

```
[ 0  2  4  6  8 10]
[0.   0.25 0.5  0.75 1.  ]
```

2.2 Properties of Numpy Arrays

- Shape : The dimentions of the array ,accessable via the "shape" attribute.
- Data Types : Use "dtype" to specify the data type of the array elements

Example :

In [154...
```
array_a =np.array([1,2,3], dtype=np.float32)# Explicitly set type to float
print(array_a)
```

```
[1. 2. 3.]
```

In [156...
```
array_a =np.array([1,2,3], dtype=float)
print(array_a)
```

```
[1. 2. 3.]
```

In [158...
```
array_a =np.array([1,2,3], dtype=int)
print(array_a)
```

```
[1 2 3]
```

In [162...
```
array_a =np.array([1,-2,3], dtype=bool)
print(array_a)
```

```
[ True  True  True]
```

In [164...
```
array_a =np.array([1,2,3], dtype=complex)
print(array_a)
```

```
[1.+0.j 2.+0.j 3.+0.j]
```

• Numpy Data Types:

o np.int32, np.int64, np.float32, np.float64, np.complex, np.bool_, etc

3. Array Manipulations

3.1 Reshaping and Resizing

• Reshape:

```
Changing the shape of an array without changing the data
```

Example :

```
In [194…  a=np.arange(6)
          print(a)
          b=a.reshape(2,3)# Reshaping to 2 rows, 3 columns
          print(b)
```

```
[0 1 2 3 4 5]
[[0 1 2]
 [3 4 5]]
```

- Flattening:

```
Convert multidimentional array into one-dimentional array using "flatten()" or ravel().
```

Example:

```
In [201…  b=np.array([[2,2,3],
                      [3,2,2]])
          c=b.flatten()
          print(c)
```

```
[2 2 3 3 2 2]
```

```
In [211…  b=np.array([[[2,2,3],
                       [3,2,2]],
                      [[2,2,3],
                       [2,2,3]]])
          c=b.ravel()
          print(c)
```

```
[2 2 3 3 2 2 2 2 3 2 2 3]
```

- Transposing:

```
Swap rows with columns using .T.
```

Example :

```
In [220…  b=np.array([[1,2]
                      ,[3,4]])
          transposed=b.T
          print(transposed)
```

```
[[1 3]
 [2 4]]
```

2.Resize:

```
The resize() function in NumPy changes the shape and size of an array in place. Unlike
reshape(), which only changes the view of the data, resize() can also modify the array's size,
truncating or padding with zeros as needed.
```

```
In [90]:  import numpy as np

          arr=np.array([1,2,4,5])
          resized_arr=np.resize(arr,(2,3))
          print(resized_arr)
```

```
[[1 2 4]
 [5 1 2]]
```

- if the new shape is larger ,resize() repeats the array elements to fill the new shape.
-
- if the new shape is a smaller,it truncates(delate) the data

```
In [98]:  arr.resize(3,3)
          print(arr)
```

```
[[1 2 4]
 [5 0 0]
 [0 0 0]]
```

Exapanding or Reducing Dimentions

We can add or remove dimentions using "np.expand_dims()" or "np.squeez()"

Expanding Dimentions:

This adds a new axis to the array,making it higher-dimentions

In [106…
```
arr=np.array([1,2,3])
expanded_arr=np.expand_dims(arr,axis=0) # Adds a new axis at position 0 (row-wise)
print(expanded_arr.shape)
```

```
(1, 3)
```

In [108…
```
print(expanded_arr)
```

```
[[1 2 3]]
```

Squeezing Dimentions:

In [ ]:
```
This removes dimentions with size 1
```

In [111…
```
arr=np.array([[[1,2,2,3]]])
squeezed_arr=np.squeeze(arr)
print(squeezed_arr)
```

```
[1 2 2 3]
```

In [113…
```
print(squeezed_arr.shape)
```

```
(4,)
```

5.Padding or Trimming an Array

Padding:

TO resize an array in a way that doesn't involve repeating values but instead pads the array with specific values or trims it use "np.pad()" for padding and slicing for trimming

In [147…
```
arr=np.array([[1,2],[2,3]])
#pad the array to make it 4x4 with zeros

padded_arr=np.pad(arr,pad_width=1,mode='constant',constant_values=0)
print(padded_arr)
```

```
[[0 0 0 0]
 [0 1 2 0]
 [0 2 3 0]
 [0 0 0 0]]
```

In [163…
```
arr=np.array([[1,2],[2,3]])
#pad the array to make it 4x4 with zeros

padded_arr=np.pad(arr,pad_width=2,mode='constant',constant_values=4)
print(padded_arr)
```

```
[[4 4 4 4 4 4]
 [4 4 4 4 4 4]
 [4 4 1 2 4 4]
 [4 4 2 3 4 4]
 [4 4 4 4 4 4]
 [4 4 4 4 4 4]]
```

we can customise "pad_width" and the values used for padding

2.Trimming:

Trimming an array is useally done by using 'slicing' the array to the desired size.

In [168…
```
arr=np.array([[1,2,3],[3,4,5],[6,7,8]])
#slice the array to trim it to 2x2

trimmed_arr=arr[:2,:2] # 0-2 rows and 0-2 elements
print(trimmed_arr)
```

```
[[1 2]
 [3 4]]
```

3.2 Concatenation and Stacking

• Concatenate:

```
Combine arrays along a specified axis using np.concatenate().
```

In [184…
```python
a=np.array([[2,3,4],
            [3,4,3],
           [2,3,4]])
b=np.array([[2,3,3],
            [2,4,3],
           [3,3,4]])
concated_arr=np.concatenate((a,b.T))
print(concated_arr)
```

```
[[2 3 4]
 [3 4 3]
 [2 3 4]
 [2 2 3]
 [3 4 3]
 [3 3 4]]
```

In [186…
```python
a=np.array([[2,3,4],
            [3,4,3]])
b=np.array([[2,3,3],
            [2,4,3]])
concated_arr=np.concatenate((a,b))
print(concated_arr)
```

```
[[2 3 4]
 [3 4 3]
 [2 3 3]
 [2 4 3]]
```

In [241…
```python
a=np.array([[1,2],
            [3,4]])
b=np.array([[3,4],
            [2,3]])
concatenated=np.concatenate((a,b.T),axis=0)
print(concatenated)
```

```
[[1 2]
 [3 4]
 [3 2]
 [4 3]]
```

In [253…
```python
a=np.array([[1,2],
            [3,4]])
b=np.array([[3,4],
            [2,3]])
concatenated=np.concatenate((a,b),axis=1)
print(concatenated)
```

```
[[1 2 3 4]
 [3 4 2 3]]
```

In [192…
```python
a=np.array([[1,2],
            [3,4]])
b=np.array([[3,4],
            [2,3]])
concatenated=np.concatenate((a.T,b),axis=1)
print(concatenated)
```

```
[[1 3 3 4]
 [2 4 2 3]]
```

• Stacking

```
Use np.vstack() and np.hstack() for vertical and horizontal stacking ,respectively .
```

Example:

In [263…
```python
a=np.array([[1,2],
            [3,4]])
b=np.array([[3,4],
            [2,3]])
```

In [267…
```python
vertical_stack=np.vstack((a,b))
print(vertical_stack)
```

```
[[1 2]
 [3 4]
 [3 4]
 [2 3]]
```

In [269…
```
vertical_stack=np.vstack((a,b.T))
print(vertical_stack)
```

```
[[1 2]
 [3 4]
 [3 2]
 [4 3]]
```

In [271…
```
Horizontal_stack=np.hstack((a,b))
print(Horizontal_stack)
```

```
[[1 2 3 4]
 [3 4 2 3]]
```

In [273…
```
Horizontal_stack=np.hstack((a,b.T))
print(Horizontal_stack)
```

```
[[1 2 3 2]
 [3 4 4 3]]
```

4. Indexing and Slicing

4.1 Basic Indexing

```
* Access elements using Indices.
```

Example:

In [285…
```
a=np.array([1,2,3,5])
print(a[0])
```

```
1
```

In [287…
```
a=np.array([1,2,3,5])
print(a[3])
```

```
5
```

4.2 Slicing:

```
    * Extract subarrays using slicing rechnices similar to Python lists.
```

In [291…
```
b=np.array([[2,3,4],[3,3,5]])
print(b[0,2])#--> 0--> first row-->2 elements
```

```
4
```

In [297…
```
print(b[0:2,1:3])
```

```
[[3 4]
 [3 5]]
```

4.3 Advanced Indexing

- Boolean Indexing:

```
Select elements based on conditions .
```

Example:

In [301…
```
a=np.array([1,2,3,4,5,6,7])
print(a[a>2])
```

```
[3 4 5 6 7]
```

- Fancy Indexing

```
 Access elements using an array of indices .
```

In [305…
```
indices=[0,2,4]---> indexes base ut will prints
print(a[indices])
```

```
[1 3 5]
```

5. Broadcasting

- BroadCasting allows operations on arrays of different shapes.NumPy automatically expands the smaller array to match the larger array's shape.

• Example of Broadcasting:

In [314…
```
a=np.array([[1],[2],[3]])
b=np.array([1,2,3])
c=a+b #The smaller array is broadcasted to match the shape of the larger array.
print(c)
```

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

In [316…
```
a=np.array([[1],[2],[3]])
b=np.array([1,2,3])
c=a*b #The smaller array is broadcasted to match the shape of the larger array.
print(c)
```

```
[[1 2 3]
 [2 4 6]
 [3 6 9]]
```

In [318…
```
a=np.array([[1],[2],[3]])
b=np.array([1,2,3])
c=a/b #The smaller array is broadcasted to match the shape of the larger array.
print(c)
```

```
[[1.         0.5        0.33333333]
 [2.         1.         0.66666667]
 [3.         1.5        1.        ]]
```

In [335…
```
a=np.array([[1],[2],[3],[7]])
b=np.array([[1,2,3],[2,1,2],[5,6,7],[8,8,9]])
c=a+b #The smaller array is broadcasted to match the shape of the larger array.
print(c)
```

```
[[ 2  3  4]
 [ 4  3  4]
 [ 8  9 10]
 [15 15 16]]
```

• Broadcasting Rules:

- 1.If the arrays have a different number of dimentions,prepared the shape of the smaller array with ones untill they have the same number of dimentions
- 2. Compare the shapes of the two arrays element-wise,starting from the trailing dimentions.If the sizes of the dimentions are equa or one of them is 1 ,broadcasting occurs.
- 3. If the sizes are incompatiable ,an error is raised.

6. Vectorized Operations

- Vectorized opearations allows you to perform element-wise operations on entire arrays without writing explict loops.

Example :

In [340…
```
a=np.array([2,3,4])
b=np.array([3,4,5])
c=a+b
d=a*b
print(c)# element wise addition
print(d)# element wise multiplications
```

```
[5 7 9]
[ 6 12 20]
```

- Numpy's vectorised opearations are optimized for performance,making them much faster than Python loops

7. Mathematical and Statistical Functions

Numpy provides a comprehensive suite of mathematical functions:

7.1 Basic Arithmetic Operations

In [350…
```
a=np.array([2,3,5])
b=np.array([2,5,6])
print(np.add(a,b))#Addition
print(np.subtract(a,b))#substraction
print(np.multiply(a,b)) # multiplications
```

```
print(np.divide(a,b)) #division
```

```
[ 4  8 11]
[ 0 -2 -1]
[ 4 15 30]
[1.         0.6        0.83333333]
```

7.2 Statistical Functions

- Functions like np.mean(),np.median(),np.std(),np.var() and np.sum() help summurize the data.

Example:

In [381…

```
data=np.array([1,2,3,4,5])
mean=np.mean(data) #mean
std=np.std(data)# standard deviation
total=np.sum(data)# total sum
var=np.var(data)#variance
median=np.median(data) # median
print(median)
print(var)
print(mean)
print(std)
print(total)
```

```
3.0
2.0
3.0
1.4142135623730951
15
```

Standard Deviation and Varience

In [388…

```
from PIL import Image

Image.open("s2.png")
```

Out[388…

Formula for Standard Deviation:

The formula for the standard deviation σ of a population is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Where:

- σ is the standard deviation.
- N is the number of data points.
- x_i represents each value in the data set.
- μ is the mean of the data set.

For a **sample** (not the entire population), the formula is:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Where:

- s is the sample standard deviation.

In [392…

```
Image.open("s1.png")
```

Out[392...

1. **Calculate the Mean (μ)**

$$\mu = \frac{1+2+3+4+5}{5} = \frac{15}{5} = 3$$

2. **Compute Each Squared Difference from the Mean**

$$(1-3)^2 = (-2)^2 = 4$$

$$(2-3)^2 = (-1)^2 = 1$$

$$(3-3)^2 = 0^2 = 0$$

$$(4-3)^2 = 1^2 = 1$$

$$(5-3)^2 = 2^2 = 4$$

3. **Sum of Squared Differences**

$$4+1+0+1+4 = 10$$

4. **Calculate Variance**

$$\text{Variance}(\sigma^2) = \frac{10}{5} = 2$$

5. **Standard Deviation**

$$\sigma = \sqrt{2} \approx 1.4142$$

In [ ]:

7.3 Aggregation Functions

- np.min() ,np.max() and np.argmax() return the minimum and maximum values ,and index of the maximum value ,respectively

Example

In [397...
```
data=np.array([[1,2,3],[4,5,6]])
max_value=np.max(data)#maximum value
min_value=np.min(data)#minimum value
print(max_value)
print(min_value)
```

```
6
1
```

In [409...
```
data=np.array([[1,2],[3,4],[5,6]])
max_value=np.max(data)#maximum value
min_value=np.min(data)#minimum value
print(max_value)
print(min_value)
```

```
6
1
```

7.4 Universal Functions (ufuncs)

- Universal Functions(ufunc) are functions that opearate element wise on arrays.they are optimised for performance and can be handle arrays of different shapes and sizes.

```
* Commonly Used Ufuncs:
```

- Mathematical functions : np.sin(),np.cos(),np.tan(),np.exp(),np.log(),np.sqrt() etc.

- Comparision functions : np.greater() ,np.less(),np.equal()..etc

In [428…
```
a=np.array([0,np.pi/2,np.pi])
print(a)
sin_value=np.sin(a) #sine values
exp_values=np.exp(a) #Exponential values
log_values=np.log(a+1) # Natural logirithm (avoid log(0))
print(sin_value)
print(log_values)
print(exp_values)
print(np.tan(a))
```

```
[0.         1.57079633 3.14159265]
[0.0000000e+00 1.0000000e+00 1.2246468e-16]
[0.         0.94421571 1.42108041]
[ 1.          4.81047738 23.14069263]
[ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

8. Linear Algebra with Numpy

```
Numpy provides a comprehensive set of linear alegebra functions via numpy.linalg module
```

8.1 Matrix Operations

- Matrix Multiplications: Use np.dot() or the @ opearator for matrix multiplication

Example

In [436…
```
A=np.array([[1,2],
            [3,4]])

B=np.array([[5,6],
            [6,7]])

C=np.dot(A,B) # Matrix multiplication
D= A@B # Alternative using the @ opearator
print(C)
print(D)
```

```
[[17 20]
 [39 46]]
[[17 20]
 [39 46]]
```

In [438…
```
Image.open("S3.png")
```

Out[438…

Summary of the Multiplication Process:

$$C = A \times B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

$$= \begin{bmatrix} (1 \times 7 + 2 \times 9 + 3 \times 11) & (1 \times 8 + 2 \times 10 + 3 \times 12) \\ (4 \times 7 + 5 \times 9 + 6 \times 11) & (4 \times 8 + 5 \times 10 + 6 \times 12) \end{bmatrix}$$

$$= \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

8.2 Determinants and Inverses

- Determinant : Calculate the determinant of a matrix using np.linalg.det().
- Inverse : Find the inverse of a matrix using np.linalg.inv()

Example:

In [452…

```python
A=np.array([[1,2],
            [3,4]])

B=np.array([[5,6],
            [6,7]])

det_A =np.linalg.det(A) #Determinant of A
det_B =np.linalg.inv(A) # Inverse of A
print(det_A)
print(det_B)
```

```
-2.0000000000000004
[[-2.   1. ]
 [ 1.5 -0.5]]
```

In [448…

```python
Image.open("s4.png")
```

Out[448…

1. Determinant of a 2x2 Matrix

For a matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The determinant is:

$$\det(A) = ad - bc$$

2. Determinant of a 3x3 Matrix

For a matrix:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

The determinant is:

$$\det(A) = a(ei - fh) - b(di - fg) + c(dh - eg)$$

In [450…

```python
Image.open("s5.png")
```

Out[450…

1. Inverse of a 2x2 Matrix

For a matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

If $\det(A) \neq 0$, the inverse is:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Example:

Given:

$$A = \begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}$$

Calculation:

$$\det(A) = (4 \times 6) - (7 \times 2) = 24 - 14 = 10$$
$$A^{-1} = \frac{1}{10} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix} = \begin{bmatrix} 0.6 & -0.7 \\ -0.2 & 0.4 \end{bmatrix}$$

8.3 Eigenvalues and Eigenvectors

Eigenvalues:

- Defination : Scalars that provide information about the behaviour of linear transformation represented by a matrix.
- Interpretation : When a matrix is multiplied by an eigenvalues ,the result is simply the eigenvector scaled by the eigenvalue.
- Example : If A is a matrix and λ is an eigenvaluemthe equation is : Av=λv

Here,v is the eigenvector

• Eigenvectors:

- Definition: Non-zero vectors that only change by a scalar factor when a linear transformation is applied to them.
- Interpretation: They indicate the direction in which the transformation acts.
- Example: In the above equation, v is an eigenvector corresponding to the eigenvalue λ.

Finding the eigenvalues and eigenvectors:

```
* Charasterstics Polynomial: To find the eigenvalues ,solve the characterstic polynomial given by "

  det(A−λI)=0
```

where I is the identity matrix.

In [10]:

```python
import numpy as np
#deifine a matrix
A=np.array([[4,2],
            [1,3]])
#calculate eigenvalues and eigenvectors
eigenvalues,eigenvectors=np.linalg.eig(A)
print("Eigenvalues:",eigenvalues)
print("Eigenvectors:",eigenvectors)
```

```
Eigenvalues: [5. 2.]
Eigenvectors: [[ 0.89442719 -0.70710678]
 [ 0.4472136   0.70710678]]
```

Intuition Behind Eigenvalues and Eigenvectors:

• Visual Representation:

- Imagine applying a transformation representation by a matrix A to a vector.Most vectors will change direction and length.However,eigenvectors are special : They only get streched or compressed by the eigenvalue factor without changing their direction.

• Applications:

- Eigenvalues and eigenvectors have significant applications in various fields ,including:
- Principle Component Analysis(PCA) in machine learning for dimentionility reduction.
- Stability analysis in systems of differential equations.

- Calculate eigenvalues and eigenvectors using np.linalg.eig()

Example

In [14]:
```
eigenvalues,eigenvectors=np.linalg.eig(A)
print("Eigenvalues:",eigenvalues)
print("Eigenvectors:",eigenvectors)
```

```
Eigenvalues: [5. 2.]
Eigenvectors: [[ 0.89442719 -0.70710678]
 [ 0.4472136   0.70710678]]
```

In [221…
```
from PIL import Image
Image.open('eigen1.jpg')
```

Out[221…

Step 1: Set Up the Eigenvalue Problem

We want to find eigenvalues λ and eigenvectors $\mathbf{v}$ that satisfy the equation:

$$A\mathbf{v} = \lambda\mathbf{v}$$

Step 2: Find the Eigenvalues

The eigenvalues are solutions to the characteristic equation:

$$\det(A - \lambda I) = 0$$

Where I is the identity matrix.

2.1: Form the Matrix $A - \lambda I$

For the given matrix $A = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$, subtract λI from A:

$$A - \lambda I = \begin{bmatrix} 4-\lambda & 2 \\ 1 & 3-\lambda \end{bmatrix}$$

2.2: Set Up the Determinant Equation

Now, compute the determinant of $A - \lambda I$:

$$\det(A - \lambda I) = \det \begin{bmatrix} 4-\lambda & 2 \\ 1 & 3-\lambda \end{bmatrix}$$

The determinant of a 2x2 matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is de $= ad - bc$. Applying this formula:

In [223…
```
from PIL import Image
Image.open('eigen2.jpg')
```

Out[223…

The determinant of a 2x2 matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is $\det = ad - bc$. Applying this formula:

$$\det(A - \lambda I) = (4-\lambda)(3-\lambda) - (1)(2)$$

Simplify:

$$\det(A - \lambda I) = (4-\lambda)(3-\lambda) - 2$$

Expand the product:

$$(4-\lambda)(3-\lambda) = 12 - 4\lambda - 3\lambda + \lambda^2 = \lambda^2 - 7\lambda + 12$$

Now subtract 2:

$$\lambda^2 - 7\lambda + 12 - 2 = \lambda^2 - 7\lambda + 10 = 0$$

2.3: Solve the Characteristic Equation

We now solve the quadratic equation:

$$\lambda^2 - 7\lambda + 10 = 0$$

Using the quadratic formula:

$$\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In [225…

```
from PIL import Image
Image.open('eigen3.jpg')
```

Out[225…

Thus, the two eigenvalues are:

$$\lambda_1 = \frac{7+3}{2} = 5, \quad \lambda_2 = \frac{7-3}{2} = 2$$

Step 3: Find the Eigenvectors

For each eigenvalue, we now find the corresponding eigenvector by solving $(A - \lambda I)\mathbf{v} = 0$.

3.1: Eigenvector for $\lambda_1 = 5$

Substitute $\lambda_1 = 5$ into $A - \lambda I$:

$$A - 5I = \begin{bmatrix} 4-5 & 2 \\ 1 & 3-5 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix}$$

Now solve $(A - 5I)\mathbf{v} = 0$:

$$\begin{bmatrix} -1 & 2 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This gives two equations:

1. $-v_1 + 2v_2 = 0$
2. $v_1 - 2v_2 = 0$

Both equations are the same, so we can choose $v_1 = 2v_2$. For simplicity, let $v_2 = 1$, then $v_1 = 2$.

Thus, the eigenvector corresponding to $\lambda_1 = 5$ is:

In [227…

```
from PIL import Image
Image.open('eigen4.jpg')
```

Out[227...

3.2: Eigenvector for $\lambda_2 = 2$

Substitute $\lambda_2 = 2$ into $A - 2I$:

$$A - 2I = \begin{bmatrix} 4-2 & 2 \\ 1 & 3-2 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix}$$

Now solve $(A - 2I)\mathbf{v} = 0$:

$$\begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This gives two equations:

1. $2v_1 + 2v_2 = 0$
2. $v_1 + v_2 = 0$

From the second equation, $v_1 = -v_2$. Let $v_2 = 1$, then $v_1 = -1$.

Thus, the eigenvector corresponding to $\lambda_2 = 2$ is:

$$\mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Step 4: Final Result

The eigenvalues and their corresponding eigenvectors for the matrix $A = \begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$ are:

- Eigenvalue $\lambda_1 = 5$ with eigenvector $\mathbf{v}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$.
- Eigenvalue $\lambda_2 = 2$ with eigenvector $\mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$.

8.4 Solving Linear Systems

- Use np.linalg.solve() to solve a system of linear equations represented as Ax=bAx=bAx=b

Example :

In [25]:

```
A=np.array([[2,3],
            [5,4]])

b=np.array([8,
            14])
x=np.linalg.solve(A,b)
print(x)
```

```
[1.42857143 1.71428571]
```

In [29]:

```
from  PIL import Image
Image.open("s7.png")
```

Out[29]:

2. **Matrix Form**:

   Using matrices simplifies the representation, especially for larger systems.

$$A\mathbf{x} = \mathbf{b}$$

- A: Coefficient matrix ($m \times n$)
- $\mathbf{x}$: Variable vector ($n \times 1$)
- $\mathbf{b}$: Constants vector ($m \times 1$)

For the earlier example:

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 4 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 8 \\ 14 \end{bmatrix}$$

In [33]:

```
Image.open("s6.png")
```

Out[33]:

Example 1: Unique Solution (Consistent and Independent)

Solve the system:

$$\begin{cases} 2x + 3y = 8 \\ 5x + 4y = 14 \end{cases}$$

Solution:

Using substitution or elimination:

1. Multiply the first equation by 5 and the second by 2 to eliminate x:

$$\begin{cases} 10x + 15y = 40 \\ 10x + 8y = 28 \end{cases}$$

2. Subtract the second equation from the first:

$$7y = 12 \implies y = \frac{12}{7}$$

3. Substitute y back into the first original equation:

$$2x + 3\left(\frac{12}{7}\right) = 8 \implies 2x = 8 - \frac{36}{7} = \frac{20}{7} \implies x = \frac{10}{7}$$

9. Random Number Generation

```
Numpy "random" module provides various functions for generating random numbers,making it
essential for simulations and testing.
```

9.1 Generating Random Numbers

• Uniform Distribution:

```
np.random.rand() generates numbers uniformly distributed between [0,1]
```

Example:

In [64]:
```
uniform_randoms=np.random.rand(3,2)# 3x2 array of random numbers
print(uniform_randoms)
```

```
[[0.98914688 0.86214457]
 [0.9530316  0.98627537]
 [0.31158399 0.76746173]]
```

9.2 Random Integers:

```
* Generate random integers with np.random.randint(low,high,size).
```

In [86]:
```
random_integers=np.random.randint(1,10,size=(2,3)) #2x3 arrays of random integers between 1 and 10
print(random_integers)
```

```
[[3 5 7]
 [7 8 7]]
```

9.3 Random Sampling

```
* Randomly shuffle an array using np.random.shuffle().
```

Example:

In [29]:
```
import numpy as np
data=np.array([1,2,3,4,5])
np.random.shuffle(data) #Shuffle the array in-place
print(data)
```

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

In [60]:
```
import numpy as np
data=np.array([[1,2],[3,4],[5,6]])
np.random.shuffle(data) #Shuffle the array in-place
print(data)
```

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

Creating a Shuffled Copy Without Modifying the Original Array:

In [44]:
```
import numpy as np

data = np.array([1, 2, 3, 4, 5])
shuffled_data = np.random.permutation(data)

print("Original array:", data)
print("Shuffled copy:", shuffled_data)
```

```
Original array: [1 2 3 4 5]
Shuffled copy: [4 1 3 2 5]
```

In [5]:
```
import numpy as np

data = np.array([[1, 2], [3, 4], [5,6]])
shuffled_data = np.random.permutation(data)

print("Original array:", data)
print("Shuffled copy:", shuffled_data)
```

```
Original array: [[1 2]
 [3 4]
 [5 6]]
Shuffled copy: [[5 6]
 [3 4]
 [1 2]]
```

Advanced Array Manipulation

10.1 Broadcasting in More Detail

- Broadcasting allows operations on arrays of different shapes.It works by expanding the smaller array to match the shape of the larger array for element-wise opearations.

Example of Broadcasting with Higher Dimentions:

In [7]:
```
A=np.array([[1,2,3],[4,5,6]]) # shape of (2,3)
B=np.array([1,2,3]) # shape(3,)
C=A+B
print(C)
#broadcasted to shape (2,3)
```

```
[[2 4 6]
 [5 7 9]]
```

10.2 Stacking and Splitting

- Stacking arrsys using np.stack() allows for creating a dimention

Example:

In [30]:
```
a=np.array([1,2,3])
b=np.array([4,5,6])
c=np.array([2,3,4])
stacked=np.stack((a,b,c),axis=0) #Stack along a new first dimention
print(stacked) #  axis=0 along stacking in horizontal way axis=1 along vertical way
```

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

In [32]:
```
a=np.array([1,2,3])
b=np.array([4,5,6])
c=np.array([2,3,4])
stacked=np.stack((a,b,c),axis=1) #Stack along a new first dimention
print(stacked) #  axis=1 along vertical way
```

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

* Splitting :

- arrays using np.split() divides an arrays into multiple sub-arrays

Example :

In [37]:
```
x=np.array([1,2,3,4,5,6])
y=np.split(x,3) # Split into 3 parts
```

In [39]:
```
print(y)
```

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

In [47]:
```
x=np.array([[1,2],[3,4],[5,6]])
y=np.split(x,3) # Split into 3 parts
```

In [43]:
```
print(y)
```

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

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: